tigerjython

Jarka Arnold, Tobias Kohn, Aegidius Plüss

PROGRAMMING CONCEPTS

in Python with the IDE TigerJython

These pages constitute an interactive and fully online textbook on programming with Python. It offers a wealth of complete code examples, exercises, and ideas on how to bring current topics to your programming class. Its primary objective is to build up a solid understanding of the concepts of programming, rather than a mere introduction to the programming language Python.

www.programmierkonzepte.ch/engl

Version 2.7, Last update: Nov 24 2021

Content

Version 2.6, July 2016

Autors: Jarka Arnold, Tobias Kohn, Aegidius Plüss
English translation: Kristin and Florian Thalmann
Contact: help@tigerjython.com

Supported by the SVIA / SSIE / SVIA
Swiss Association for computer science in education

FOREWORD

In the beginning of the 1950s I had the privilege of using the first available programmable computer in Switzerland, Zuse 4, to write my doctoral thesis at the Swiss Federal Institute of Technology (ETHZ) in Zürich. The first steps in our country in computer sciences, later combined under the name "informatics", only gradually found their way into cantonal universities, particularly the recognition of informatics as a separate scientific discipline. At the ETHZ it was only in 1974 that the group of computer science professors got their own institute in computer science and it took until 1981 for an entire department of computer science to be initiated. The rapid development of the performance and the miniaturization of computers significantly contributed to a huge increase in data production. This led to an enormous expansion of data communications, which could only be accomplished with a more widespread use of computers. Consequently, communication techniques had to be expanded and access to them had to be enabled.

On our planet, which is threatened by an increasing growth of populations and their demands for better living conditions, Switzerland can only maintain its position as one of the wealthiest and most advanced countries with an exquisite standard of living, and direct democracy by having a modern and efficient education system with high quality research. However, this requires not only taking account of the latest developments in the ICT sector in an optimal expansion of our universities, but also a redesign of the basic education offered in primary and secondary schools, as well as the teaching of computer science in grammar schools. The three basic skills of reading, writing, and arithmetic are no longer sufficient to ensure a satisfactory existence in today's world, where computers play a key role in personal and professional life. As the former Director of the Federal Office for Education and Science, I fought for establishing computer science as a separate subject in grammar schools.

A question that arose was whether the integration of ICT subjects in teaching programs was sufficient enough, or whether a more comprehensive knowledge in computer science, which allows for a better use of modern computer technology, should be taught. The authors of the learning platform TigerJython, who demonstrate how the most important concepts in computer science can be taught in a simple way while using the Python programming language and a didactically designed programming environment, give a concrete answer to this question. It provides an excellent foundation for their recommendation that the subject of computer science be introduced in the 6th grade. The following recently published Swiss press release titled "Die Schweizer EGovernmentAngebote sind im internationalen Vergleich nur Mittelmass... Die Schweiz ist unter den europäischen Staaten gar auf den vorletzten Platz zurückgefallen" shows, in my opinion, the need for a timely response to this proposal since this alarming regression is above all due to an inadequate knowledge of computer science in the educational institutions. In our rich country with its high density of computers there exists no lack of the necessary material requirements to correct this relaps!

Prof. Dr. sc. math., Dr. h.c. Urs Hochstrasser, former Director of the Federal Office for Education and Science (**http://hochstrasserurs.blogspot.ch**)

PREFACE

TigerJython consists of online teaching materials and a development environment specially designed for education. The online teaching materials begin with turtle graphics, but then continue on with topics reaching from the programming of Lego robots, multimedia, and computer games to databases and stochastic simulations. *TigerJython* is suitable for use both in the classroom and for self-study due to its modular structure and its numerous examples and exercises. The first chapters can already be used in introductory computer science courses in elementary schools (in Switzerland S1). As a whole, the choice of topics and the material's scope correspond to a basic course of computer science at grammar schools.

The authors are convinced that education in computer science contributes essentially to the intellectual development of adolescents. In our opinion, it should already be taught in primary schools no later than at the age of 12-13 years old, in order to awaken an early enjoyment and interest in logical and technical thinking in pupils. The first version of the teaching materials were developed in 2013, and this is now the second version with revised and corrected materials. Throughout these materials we have incorporated our years of experience with students, as well as our experience in educating computer science teaching staff. Our intention was always to develop the interest and enjoyment of algorithmic problem solving in girls and boys, and to support teachers.

In this teaching material, any barriers that might prevent someone from entering into programming are deliberately kept very low, and throughout, the *TigerJython* programming environment and the *Python* programming language are used. The teaching material was formed from a single mold, so to speak. Much of the content comes from daily environments and problem situations in other school subjects. This way, the knowledge from computer science class can be applied to other disciplines as well.

Although *Python* was developed by Dutchman Guido van Rossum already over 20 years ago, it has only really become used in schools in recent years and is now experiencing a real 'hype' in many training institutions. This may be because *Python* as an interpreted language with its global scope is very easy to learn, but also because *Python* can work with very few computer resources and even runs on micro systems. Also, with our development environment *TigerJython* we offer a student-friendly environment that is balanced between simplicity and professionalism. In our opinion, it is particularly well suited for a computer science class for the following reasons:

- Installation on Windows/Mac/Linux consists of copying a single file to the computer. This way, instructors can immediately start teaching, even in computer labs without administrator rights
- The IDE is so simple that absolutely no introductory instruction is necessary for its operation. Particularly, there is no need to create projects
- *TigerJython* performs a precise error analysis of the program, and outputs error messages that are understandable by novice programmers
- *TigerJython* contains numerous additional modules that are specifically tailored for the classroom, such as turtle graphics, coordinate graphics, robotics, and game programming

We hope that with *TigerJython* and the online teaching materials we are able to pass on some of our enthusiasm for education in computer science.

Acknowledgements:

We want to thank all of those who contributed to the success of *TigerJython* with suggestions and feedback, namely Walter Gander (ETH Zürich), Juraj Hromkovic (ETH Zürich), Theo Heußer (Gymnasium Hemsbach), Urs Hochstrasser (former Federal Office for Education and Science, Bern).

October 2014. Jarka Arnold, Tobias Kohn, Aegidius Plüss

chapter one

LEARNING ENVIRONMENT

Learning Objectives

- You can install the TigerJython development environment on your computer.
- You know how to edit and running a program.
- You know how to change settings.
- You know how to use the console window for simple calculations.
- You know that you can even use TigerJython on the Raspberry PI.

"I think everybody in this country should learn how to program a computer because it teaches you how to think."

Steve Jobs, The lost interview

1.1 SETUP

INTRODUCTION

The development environment of TigerJython is well suited for novice programmers and for users who work in a protected environment (for example, computer labs without administrator rights). The distribution of TigerJython consists of a single JAR file that can be downloaded for free.

Download TigerJython

The distribution contains all of the components necessary for programming, except for the Java Runtime Environment (**JRE**). TigerJython is even capable of running from an external data storage device (USB-Stick, CD).

TigerJython is an **independent platform** that works flawlessly with Windows, Mac and Linux, and even Raspberry Pi.

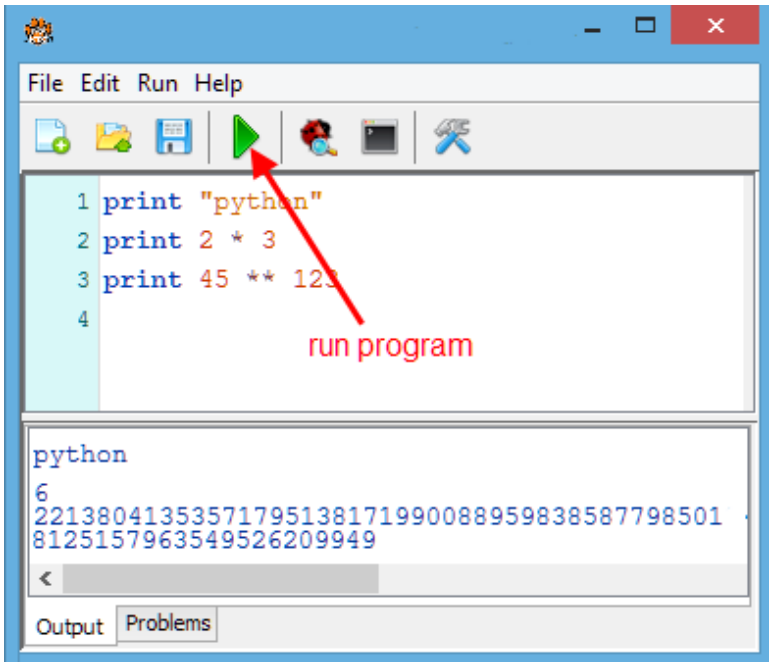
INSTALLATION

Download the file *tigerjython2.jar*. Save it to any directory on your hard drive, and if you would like to, you can create a linked file so that you can start it directly from the desktop. If you are on Linux, you have to give the JAR file the right to run (*executable*). You can save all of your Python programs in the same directory. If you wish to assign a corresponding desktop icon to the link, you can download it **here** for Windows and **here** for Mac/Linux.

GETTING STARTED

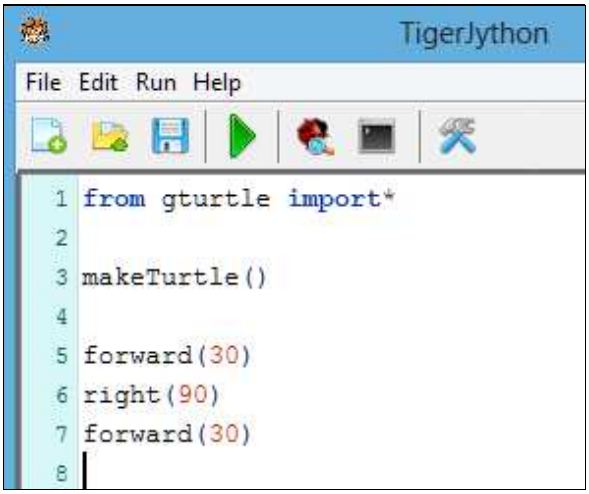
Start the TigerJython-Editor either by clicking on ***tigerjython2.jar*** or by clicking on the link to the file.

The editor is easy to operate. There are buttons for *New document, Open, Save, Run program, Debugger (on/off), Display console* and *Settings.* Test it out by typing in some print-commands, then click on the green *Run program* button. Unlike most other programming languages, Python can deal with numbers of any length.



EDITING PROGRAM

Write a simple program to create a turtle graphic



When editing, you can use standard keyboard shortcuts:

Ctrl+C	Copy
Ctrl+V	Paste
Ctrl+X	Cut
Ctrl+A	Select all
Ctrl+Z	Undo
Ctrl+S	Save
Ctrl+N	New document
Ctrl+O	Open
Ctrl+Y	Redo
Ctrl+F	Search
Ctrl+H	Search and Replace
Ctrl+Q	Comment out selected lines Remove comment
Ctrl+D	Delete row
Shift+ Cursor	Highlight

```
from gturtle import *

makeTurtle()

forward(141)
left(135)
forward(100)
left(90)
forward(100)
```

Highlight program code (Ctrl+C copy, Ctrl+V

The example programs used in the tutorial are chosen so that you are able to use them easily as templates.

You can select the entire program by clicking on *highlight program code*. You can also select a part of the code by using the mouse. Use Ctrl+C to copy the highlighted code to the clipboard and Ctrl+V to paste it into your TigerJython-editor window.

With the **import** command you tell the computer that it should make certain commands in a module available. The command **makeTurtle()** creates a window with a turtle that you can control. The following lines of the code consist of commands (also called statements) for the turtle itself.

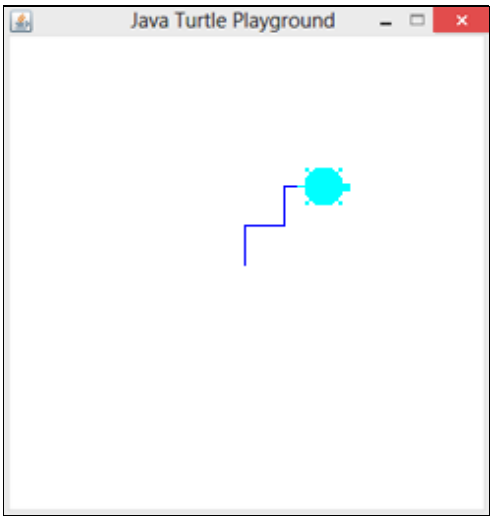
```
from gturtle import *

makeTurtle()

forward(141)
left(135)
forward(100)
left(90)
forward(100)
```

Highlight program code (Ctrl+C copy, Ctrl+Vaste)

Our **highlighting trick** will help you find the statements that are mentioned in the text, in the program.

By clicking on words written in green, the corresponding statement is highlighted in the program.

RUNNING THE PROGRAM



Click on the green arrow to run the program.

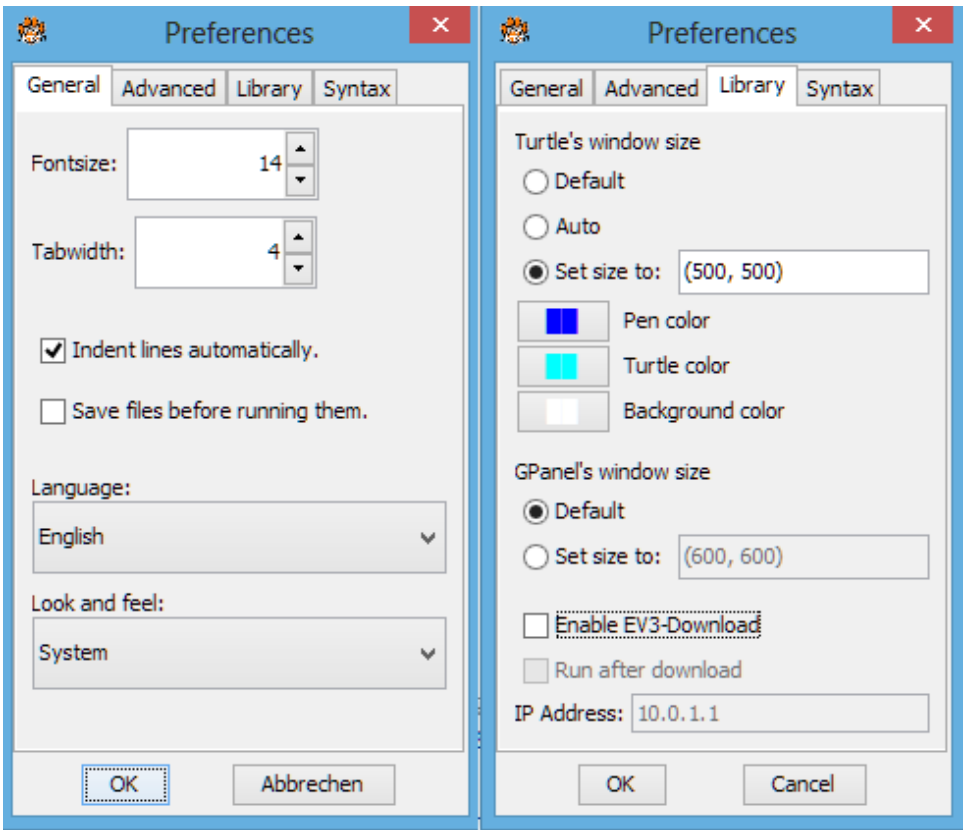
The graphic appears in a new window..

If something is wrong with the program, error messages will appear in the *problems* window

SETTINGS

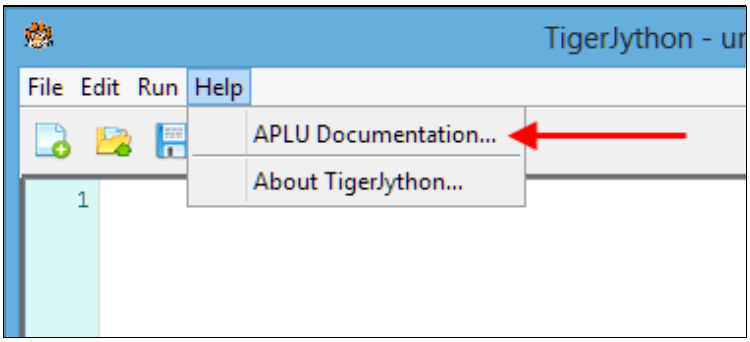
You can make some adjustments under the settings:

- Font size, indentation and font colors of the editor
- Language (German, English, French)
- Default size and background color of the turtle window, pen and turtle color
- Additional tools for enabling EV3-robotics, etc.



DOCUMENTATION

There are additional modules that are integrated into TigerJython, for example the turtle graphic. By clicking on the *APLU documentation* in the *Help* tab, you can view the documentation for these libraries.



EXAMPLES

We suggest that you work through the teaching material chapter by chapter. Transfer each example program individually to the TigerJython editor using *highlight program code, Ctrl+C* and *Ctrl+V,* save them with an appropriate name, and then execute them.

You can also **download all of the programs here**.

INSTALLATION IN COMPUTER LABS FOR MULTIPLE USERS

TigerJython is limited to a single JAR file *tigerjython2.jar*, so it is easily removable from any computer. No installation process is required and no registry entries are made. For user-definable options, a configuration file named *tigerjython2.cfg* is used, which is usually automatically generated in the home directory of *tigerjython2.jar*. In computer labs, this file can be managed by a system administrator. More information can be found **here**.

Note: In rare cases in computer labs, the JARs of APLU libraries (e.g. aplu5.jar) used by Java are copied into *<jrehome>/lib/ext*. This may lead to conflicts with TigerJython which uses specifically configured APLU libraries.

DESKTOP LAUNCHER FOR UBUNTU

Download image file tjlogo64.png from **here** and copy it into the directory where is *tigerjython2.jar*.

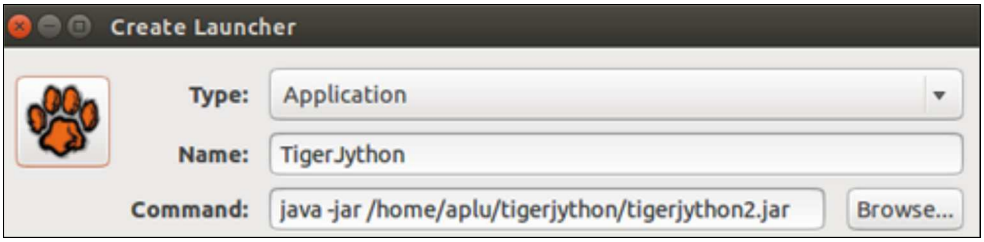
For newer versions of Ubuntu, the gnome-panel must be installed:

```
sudo apt-get install gnome-panel
```

Generating the launcher file by pressing Alt-F2 and entering

```
gnome-desktop-item-edit --create-new ~/Desktop
```

The dialog box fill (path to adjust *tigerjython2.jar*):



Click on the icone and specify the downloaded image file *tjlogo64.png*. Confirm with OK.

STARTING A PROGRAM WITHOUT THE TIGERJYTHON IDE

Since Python is an interpreted language, it is necessary that the interpreter is started to execute a program script. Under Windows you may run a script from the command line with

java -jar jython.jar <prog.py>

provided that the current directory contains jython.jar and the script.

To ensure that the additional modules from the APLU library are automatically loaded, they have to be included in the JAR file. **Here** you can download a modified jython.jar (named ajython.jar) that contains the modules. Check the *readme.txt* in the download to get more information how to proceed. Be aware that some TigerJython specific language features are missing, especially the repeat structure and some input dialogs. However it is not necessary to install Python nor Jython.

1.2 FIRST STEPS

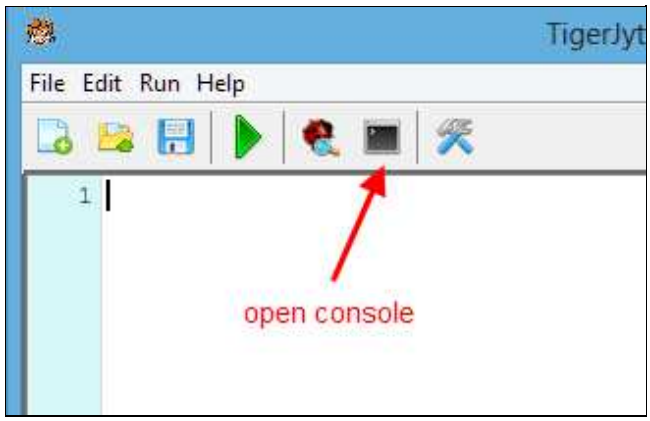
INTRODUCTION

A computer program typically consists of several statements. With Python, you can immediately execute single statements. This approach is a particularly good way to try out Python for the first time or test something out. To get started you have to click on the console icon, which opens the console window. On the command line that starts with >>>, you can type the instruction and then end it with the ENTER key (carriage return). As in any other ordinary editor, you can easily use the cursor keys to move back and forth on the command line and delete or insert single characters. As soon as you press ENTER the command line is executed, unless it is a multi-line command. In this case, the command is only executed after you press ENTER repeatedly.

You can mark already processed statements with the mouse, and copy it to the clipboard using *Ctrl+C*. You can paste the contents of the clipboard when you are on the command line using *Ctrl+V*.

The *underline* symbol is a placeholder for the result of a previous calculation, using *Cursor-Up* you can get the last command and using Cursor-Left/Cursor-Right edit.

GETTING TO KNOW PYTHON



You can vary the following proposals as you wish, how ever you might find them to be more interesting or fun. Start TigerJython and select the *Console* button.

Start by typing the examples below to get to know the four basic arithmetic operations:

```
>>> 4 + 13
    17

>>> 2.5 - 5.7
    -3.2

>>> 1356 * 22345
    30299820

>>> 1 / 7
    0.14285714285714285
```

As you see, you can use whole numbers or decimals. The whole numbers are called *integer (int)* and the decimals are called *float*.

You can write several operations on a single line. Pay attention to the **order of operations** that applies, where * and / bind more strongly than + and -, and with operations of the same rank, the expression is evaluated from left to right. You can put the operations that belong together in parentheses. (Square and curly brackets have different meanings):

```
>>> (66 - 12) * 5 / 6
    45.0

>>> 66 - 12 * 5 / 6
    56.0
```

The **integer division** and the **remainder** (modulo operation) are also important:

```
>>> 5 // 3
    1

>>> 5 % 3
    2
```

Python can easily manage long numbers without a problem, for example with the use of the **power operator**:

```
>>> 45**123
    221380413535717951381719900889598385877985018125157 96
    354952620994941135358805405606080888944357204960582 62
    034077378666827289015081270841515229492687489761281 37
    612813664505432287299413474102038890123367309570312 5L
```

There are a number of built-in functions, for example:

```
>>> abs(-9)
    9

>>> max(1, 5, 2, 4)
    5
```

Many other functions are available only after you import the respective modules. You can import in two ways. In the first way, you must precede a function with its module name followed by a dot. In the second way, you can call the function directly.

```
>>> import math
>>> math.pi
    3.141592653589793
>>> math.cos(pi)
    -1
```

```
>>> from math import *
>>> pi
    3.141592653589793
>>> sin(pi)
    1.2246467991473533e-16
```

Here you can see that a computer program never calculates exactly, since *sin(pi)* would have to be exactly 0.

A succession of letters and punctuation marks is called ***string*** and you can define it by using single or double quotes. With the *print* command, you can write strings and other values to an output window. The comma is used as a separator.

```
>>> print "The result is", 2
```

Produces in the output window: The result is 2

As in mathematics, you can assign values to variable names. To do this, use an identifier of one or more letters. Some characters are not allowed such as blank spaces, umlauts, accents and most other special characters. One benefit of using variables is to achieve a previously calculated result faster. Quite conveniently, the already known variables are listed in the right section of the console window.

```
>>> a = 2
>>> b = 3
>>> sum = a + b
>>> print "The sum of", a, "and", b, "is", sum
```

Produces in the output window: : `The Sum von 2 and 3 is 5`

A one-dimensional collection of arbitrary data is called a ***list***. Lists are a highly convenient and flexible data type in all programming languages. In Python you simply write list items in square brackets and you can also display them in the output window by calling *print*.

```
>>> li = [1, "chicken", 3.14]
>>> print li
```

In the output window: `[1, "chicken", 3.14]`

Lists and many other objects have associated functions which are called *methods*. For example, you can add a new element to the end of the list with the method ***append()***.

```
>>> li.append("egg")
>>> print li
```

In the output window: `[1, 'chicken', 3.14, 'egg']`

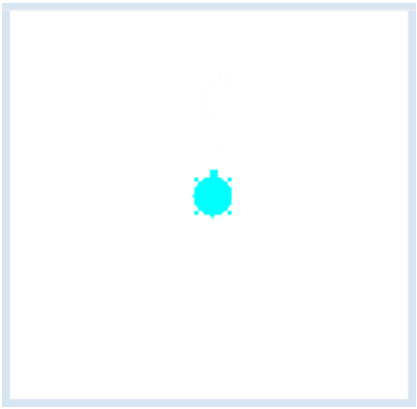
You can also **define your own functions.** For this purpose, you will use the keyword *def.* You can return values using *return*. After you define it, you can call your function as you would with any other built-in function:

```
>>> def sum(a, b):
>>>     return a + b
>>> sum(2, 3)

   5
```

GIVING THE TURTLE COMMANDS

The console is very useful for quickly trying out a few commands or functions. For example, if you want to familiarize yourself with turtle graphics, first import the module *gturtle* and then create a window with a turtle in it using the command *makeTurtle().*

```
>>> from gturtle import *
>>> makeTurtle()
```

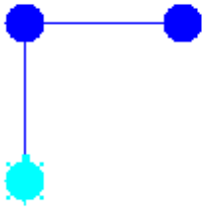


Afterwards, you have all the commands of turtle graphics at your disposal. For example:

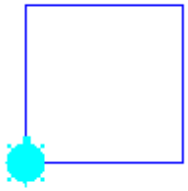
forward(100)	short: fd(100)	Move 100 steps (pixels) forward
back(50)	short: bk(50)	Move 50 steps backwards
left(90)	short lt(90)	Rotate 90° to the left
right(90)	short: rt(90)	Rotate 90° to the right
clearScreen()	short: cs()	Delete all traces and place the turtle in the middle

Example:

```
>>> fd(100)
>>> dot(20)
>>> rt(90)
>>> fd(100)
>>> dot(20)
>>> home()
```

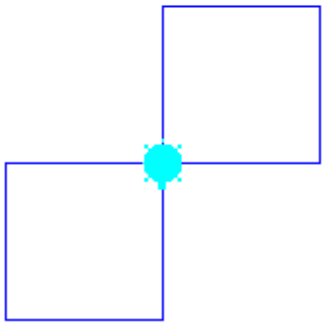


With the keyword *repeat* you can execute one or more statements repeatedly. If you want to repeat a series of commands as a command block, you have to indent them by the same amount.



```
>>> repeat 4:
       fd(100)
       rt(90)
```

As shown above, you can combine several statements under their own name by defining your own function. The main advantage of functions is that you can call them by their name as often as you would like, instead of writing down their code in its entirety each time.



```
>>> def drawSquare():
      repeat 4:
         fd(100)
         rt(90)

>>> drawSquare()
>>> rt(180)
>>> drawSquare()
```

It might be fun to try some more turtle commands on your own. You can find an overview of the commands in the chapter Turtle Graphics under **dokumentation**. In that chapter you will also systematically learn how to write entire programs.

1.3 INSTRUCTIONS FOR TEACHERS

The teaching material has an internal methodological structure that transitions "**from simple to complex**". Later chapters apply the basic knowledge and concepts that are covered in the preceding chapters. In total the material covers around 2-3 years of basic lessons. Depending on the grade of the class and the number of lessons available, only selected parts of the material can be taught, and missing terms and concepts have to made up. Since turtle graphics are a great way to be introduced to the material, most basics are taught in this chapter.

Based on our teaching experience, we suggest the following **minimal program variants.**

1. Turtle graphics and student projects (as an introduction to computer science in secondary education S1 and S2, 1-2 hours per week for a year)

2. Selected topics in turtle graphics and the chapter on robotics (as an introduction to computer science or for computer science workshops, if Lego EV3 or NXT-robots are available, at least 10-20 lessons)

3. Selected topics in turtle graphics and game programming (1-2 hours per week for a year at higher schools)

4. Selected topics in turtle graphics, coordinate graphics and applications in school subjects (as an introduction to computer science with interdisciplinary applications, 1-2 lessons per week for a year)

5. The first topics of turtle graphics (as an introduction to programming in ICT courses, 4-10 lessons)

We decided to use **English identifiers** and comments in the programs. This not only helps to facilitate the translation into other languages, but also corresponds with the trend towards the internationalization of program code.

As an aid for teachers there is a keyword list including important **concepts in computer science** for each topic.

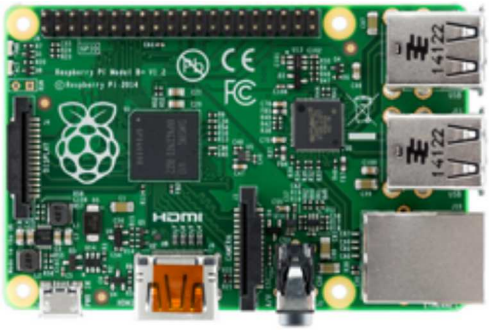
Solutions to the exercises:

If you are working at an educational institution you can get the solutions of the exercises by writing an e-mail to ***help@tigerjython.com***. The request must include the following verifiable information: name, address, educational institute, and e-mail address. By requesting the solutions, you agree that you will use them strictly for personal use and that you will not pass them on to anyone else.

Copyrights:

1.4 RASPBERRY PI

INTRODUCTION



You can use TigerJython on the Raspberry Pi in order to learn the Python programming language or in order to access its sound system and GPIO port. Although TigerJython starts slightly slower than the pre-installed Python with IDLE, you will have a more sophisticated graphical development environment with many library routines already integrated (turtle graphics, robotics, game development, etc.) at your selection.

INSTALLATION

The easiest way to get started is downloading the operating system installer NOOS from **http://www.raspberrypi.org/downloads** copying the content to a SD card (a minimum of 8 GB), and choosing the operating system *Raspbian* (a Linux Debian variant) when starting up. Since the distribution already includes a JRE, you will only need to copy *tigerjython2.jar* into a directory, for example */home/pi/tigerjython*, and give the file execution rights with the file manager (under *File properties*).

To start TigerJython, type the following command into a terminal (console):

java -jar /home/pi/tigerjython/tigerjython2.jar

In order to use the GPIO module, TigerJython requires administrator rights. That is why you should always start TigerJython with a preceding sudo:

sudo java -jar /home/pi/tigerjython/tigerjython2.jar

Instead of typing this command each and every time, you can specify in the file manager that files with the file type .jar are always executed using this command. You can also create a shell script. To make it even easier, you can make a desktop link proceeding as follows:

- Right click and copy the icon *IDLE* and then paste it onto the desktop, creating a new link icon.

- In order to edit the associated link script, right click on this icon and choose *Leafpad.* You can now adjust the entries accordingly and even specify a TigerJython logo (**download**). Here is an example:

```
Datei  Bearbeiten  Suchen  Optionen  Hilfe
[Desktop Entry]
Name=Tiger
Exec=sudo java -jar /home/pi/tigerjython/tigerjython2.jar
Icon=/usr/share/pixmaps/tjlogo1.png
Terminal=false
Type=Application
Categories=Application;Development;
StartupNotify=true
```

After saving, you can start TigerJython by clicking on the new icon. You will need a little bit of

patience until the IDE has started on the Raspberry Pi (about one minute). Fortunately, as you will notice, running Python programs is surprisingly fast. We recommend to use a fast SD card (class 10) and the Raspberry Pi 2 Model B.

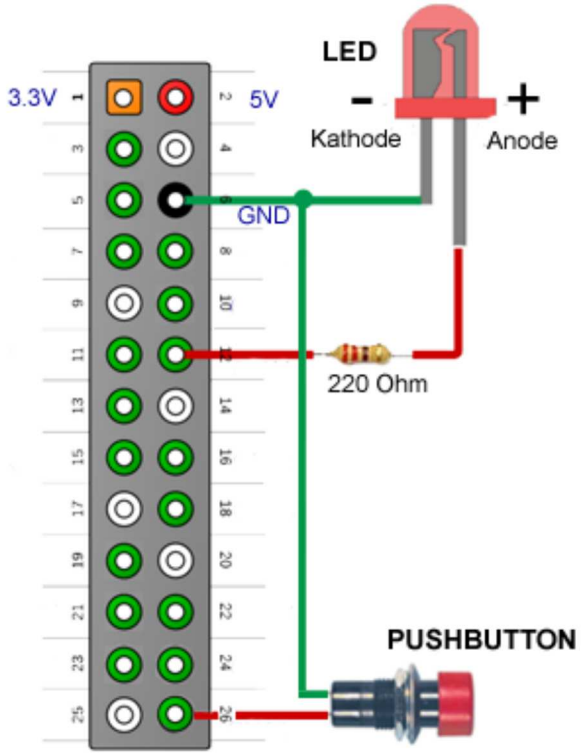
DIGITAL INPUT/OUTPUT USING THE GPIO PORT

Raspberry Pi provides you with 17 digital input/output channels that can be tapped on a 26-pin connector (new version 40-pin also supports). Each channel can be defined as an output or an input with either an internal pull-up resistor, a pull-down resistor, or without a resistor. There are also 5V, 3.3V, and ground pins available on the connectors. **The input voltage must never exceed 3.3V, so you should not connect external 5V-logic outputs directly to the inputs.**

In TigerJython, you will find the class *GPIO* in the module *RPi_GPIO*, using which you can easily address the IO ports. The module uses the library Pi4J by Robert Savage, but corresponds to the module **RPi.GPIO**. All the necessary files are included in the distribution of TigerJython.

By default, the pins of the GPIO ports are used with the pin numbers 1..26. Each pin can be defined with *GPIO.setup()* as input or output channel. With *GPIO.output(channel, state)* can you set an output value. With *GPIO.input(channel)* can you read und return the current input value. You find the detailed documentation under the menu option *Help > APLU Documentation*.

In order to test it, you simply connect an LED to a series resistance between pin 6 (ground) and pin 12, whereby you have to try out the polarity of the LED (it is not destroyed due to an incorrect polarity). If you have a key switch available, you can connect it between pin 6 (ground) and pin 26.



In your program you first define channel 12 as an output and make the LED blink 10 times per second:

```
from RPi_GPIO import GPIO

GPIO.setup(12, GPIO.OUT)

while True:
    GPIO.output(12, 1)
    GPIO.delay(100)
    GPIO.output(12, 0)
    GPIO.delay(100)
```

For the demonstration of an input port you connect a key switch at pin 26. Typically you will need a few flags, so you can turn on the blinking by pressing the button and turn it off by pressing it again.

```
from RPi_GPIO import GPIO

# pin numbers
switch = 26
led = 12

print "Press button turn blinking on/off"

GPIO.setup(led, GPIO.OUT)
GPIO.setup(switch, GPIO.IN, GPIO.PUD_UP)
```

```
buttonPressed = False
blinking = False
ledOn = False

while True:
    v = GPIO.input(switch)
    if not buttonPressed and v == GPIO.LOW:
        buttonPressed = True
        blinking = not blinking

    if buttonPressed and v == GPIO.HIGH:
        buttonPressed = False

    if blinking:
        if ledOn:
            ledOn = False
            GPIO.output(led, GPIO.LOW)
        else:
            ledOn = True
            GPIO.output(led, GPIO.HIGH)
    else:
        ledOn = False
        GPIO.output(led, GPIO.LOW)
    GPIO.delay(100)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In the loop you first get the current state of the key switch (you "poll" it). When pressed, it connects GND to the input on pin 26, therefore *input(26)* returns GPIO LOW or 0. When you release the button, it causes the internal pull-up resistor to set the input to logic HIGH (3.3V), without needing to create the voltage from the outside.

The use of the key switch as an on/off switch is a little tricky, because despite the constant polling you have to convert the turning on/off into an event that only occurs when the switch goes down. For this you use a flag *buttonPressed*, which you set to *True* upon the first key pressing. After that you do not walk through the respective part of the code again, until you have released the switch and pressed it again.

Since the loop is also responsible for the flashing, you use a flag *blinking*, that represents the on/off status. Finally, you have to remember with the flag *ledOn*, whether in a specific iteration of the loop the LED should be turned on or off.

Because you are not sure if the LED is lit up at the moment of switching off, you switch it off in the last *else*. It is a small imperfection that after that, the program will still always run through that code.

An electronic engineer knows that when a key is pressed it does not usually make immediate contact (it "bounces"). However, since the loop is only repeated after 100 ms due to the *delay(100)*, we can assume that the "bouncing" will have ended.

USING EVENTS

It is much easier to use the event model. The pressing and releasing the button is here regarded as an event that calls automatically a function *onButtonPressed()*. There, you only need the flag blinking reverse.

```
from RPi_GPIO import GPIO

def onButtonPressed(channel, state):
   global blinking
   blinking = not blinking

# pin numbers
switch = 26
led = 12

print "Press button to turn blinking on/off"
GPIO.setup(led, GPIO.OUT)
GPIO.setup(switch, GPIO.IN, GPIO.PUD_UP) # pull-up resistor
GPIO.add_event_detect(switch, GPIO.FALLING) # event on falling edge
GPIO.add_event_callback(switch, onButtonPressed) # register callback

blinking = False
while True:
    if blinking:
        GPIO.output(led, 1)
        GPIO.delay(100)
        GPIO.output(led, 0)
    GPIO.delay(100)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

To use events, you must specify with the method *add_event_detect()*, whether you want to respond to the transition from low to high or high to low or both. After that you register with *add_event_callback()* a function to be called when the event occurs.

chapter two

TURTLE GRAPHICS

Learning Objectives

- You can write a simple program and draw figures on the screen with the turtle.
- You can change the color of the lines and areas, and also adjust the line width.
- You know how the turtle can repeat statements several times.
- You know how to run parts of the program only under certain conditions.
- You can define your own commands with parameters.
- You know what variables are and you can use them in your programs.
- You know what recursion is and you can write simple recursive programs.
- You can create turtle objects and use multiple turtles simultaneously.

"A turtle is located at a certain place and it also has a certain viewing direction. Therefore, a turtle is like a person... children can identify with the turtle and can transfer their knowledge of their bodies into the learning of geometry."

Seymour Papert

2.1 MOVING THE TURTLE

INTRODUCTION

Programming means giving a machine commands in order to control it. The first such machine that you will control is a small turtle on the screen, which we simply call *turtle*. What can this turtle do and what do you have to know in order to control it?

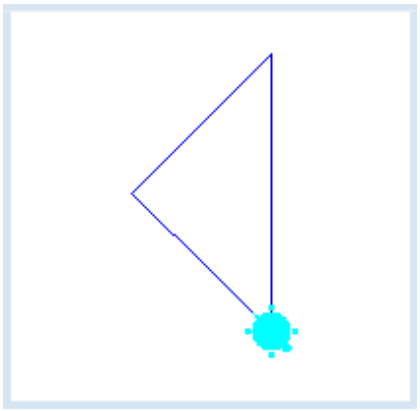
Turtle commands are written in English and are always followed by a pair of parentheses, which may contain further details about the respective command. Even if no further information is required, an empty pair of parentheses must follow. The case of the letters (either upper case or lower case) must always stay consistent.

The turtle can move within its window and draw a trail, but before it can get going, you must first instruct the computer to create such a turtle. You can do this with the command *makeTurtle()*. In order to then move the turtle, you can use three commands: *forward(distance)*, *left(angle)*, and *right(angle)*.

PROGRAMMING CONCEPTS: *Edit source, run program, sequence*

YOUR FIRST PROGRAM

This is how your first program with the turtle looks. Click on *Mark program code*, copy it and paste it into the TigerJython-Editor. Execute it by clicking on the green start button. The turtle will draw a right triangle. The turtle commands are all stored in a file (a so-called module) named *gturtle*. With the **import** command you tell the computer that it should make certain commands in a module available. The command **makeTurtle()** creates a window with a turtle that you can control. The following lines of the code consist of commands (also called statements) for the turtle itself.



```
from gturtle import *

makeTurtle()

forward(141)
left(135)
forward(100)
left(90)
forward(100)
```

MEMO

At the beginning of each turtle program you must first load the turtle module, and then create a new turtle:

```
from gturtle import *
makeTurtle()
```

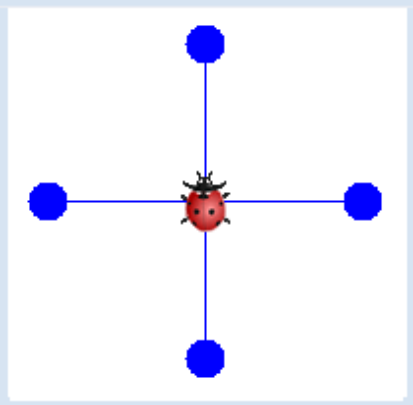
Afterwards, you can give any amount of commands to the turtle. The three commands that the turtle surely understands are:

forward(s)	Move forward by distance s (in pixels).
left(w)	Rotate left by angle w (in degrees).
right(w)	Rotate right by angle w (in degrees).

YOUR OWN TURTLE IMAGE

You can also specify your own file while calling **makeTurtle()**, which is then used as the turtle picture. This way, you can give the program your own personal touch. Here you can use the file *beetle.gif* from the directory *sprites* of the TigerJython distribution. Please note that you must put the file name in quotation marks.

With the following code, the turtle will draw a cross with filled circles at the end points.



```
from gturtle import *

makeTurtle("sprites/beetle.gif")

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)
```

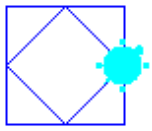
MEMO

If you want to use a different image for the turtle, as seen in the above example, you must first create an icon with an image editor. Normally, turtle images have a size of 32x32 pixels and a transparent background and are typically in GIF or PNG format. The image file should be stored in the subfolder *sprites* of the same directory in which your program is located.

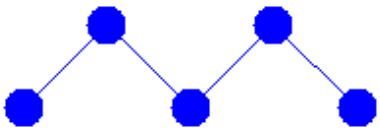
In the above program you are using the new command **back()**, with which the turtle moves backwards, as well as **dot()**, with which the turtle draws a filled circle, the radius of which you can specify (in pixels).

EXERCISES

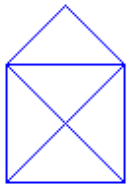
1. Draw two nested squares with the turtle.



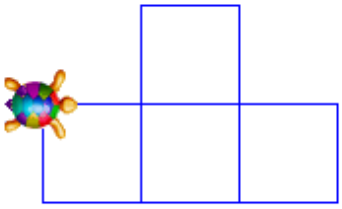
2. Using the command *dot()*, try to draw the following figure:



3. The House of Saint Nick is a drawing game for kids. The goal is to draw the house using exactly 8 lines, without passing through the same route twice. Draw the House of Saint Nick using the turtle.



4*. Create your own turtle icon with an image editor and draw the adjacent picture with it. The side length of the squares is 100. It does not matter where the turtle begins or ends.



2.2 USING COLORS

INTRODUCTION

The turtle draws its trail using a colored pen, for which it knows some additional instructions. As long as the pen is *down*, the turtle draws a trail. Using the statement *penUp()*, it moves its pen up and stops drawing. With *pen*Down(), the pen is brought back down to the drawing area, so that a trail is drawn again.

Using *setPenColor(color)* you can select the color of the pen. It is important that you put the name of the color inside quotation marks. As always in programming, the turtle knows only English color names. The following list is not complete, but here are some examples: *yellow, gold, orange, red, maroon, violet, magenta, purple, navy, blue, skyblue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white.*

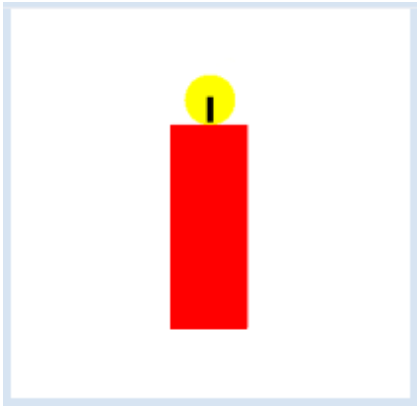
PROGRAMMING CONCEPTS: *Drawing with colors*

COLOR AND PEN WIDTH

This program makes the turtle draw a candle with a wide **red** line. You can set the line width in pixels using the command **setLineWidth()**.

You can draw the **yellow** flame with the command *dot()*. There is one part of the program where the turtle moves without drawing a line, because the pen was lifted with the command **penUp()**. After **penDown()** is called, the turtle draws again.

hideTurtle() makes the turtle invisible.



```
from gturtle import *

makeTurtle()

setLineWidth(60)
setPenColor("red")
forward(100)
penUp()
forward(50)
penDown()
setPenColor("yellow")
dot(40)
setLineWidth(5)
setPenColor("black")
back(15)
hideTurtle()
```

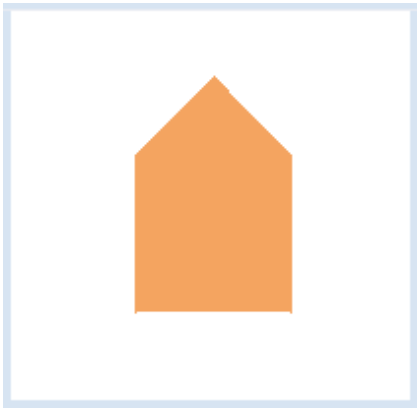
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The drawing pen of the turtle can change color with use of the statement **setPenColor(color)**. With **penUp()** the turtle stops drawing, and with **penDown()** it continues to draw again. You can control the width of the line using **setLineWidth(width)**.

The turtle knows the so-called ***X11 colors***. There are a few dozen names of colors which you can find on the Internet **http://cng.seas.rochester.edu/CNG/docs/x11color.html** . You can select all of these colors with the *setPenColor(color)* statement.

FILLED AREAS



You can fill almost any area with colors using the turtle. With the command **startPath()**, you tell the turtle that you intend to fill an area. The turtle remembers its current position as the starting point of a sequence of lines. You then move around the area with the turtle and finally call the command **fillPath()**, which connects the start point to the end point and fills in the resulting area with color. You can adjust the fill color with **setFillColor(color)**.

Lines starting with the hash symbol (#) are called **comments**, which are ignored during the execution of the program. You can add these to make notes for yourself or others.

For example, you could specify under which program name the file is stored, or add text that explains your code.

```
from gturtle import *

makeTurtle()

setPenColor("sandybrown")
setFillColor("sandybrown")
startPath()
forward(100)
right(45)
forward(72)
right(90)
forward(72)
right(45)
forward(100)
fillPath()
hideTurtle()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

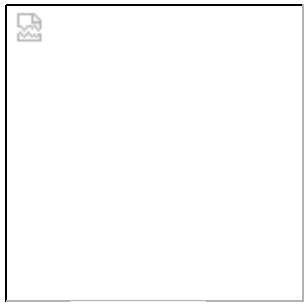
MEMO

If you want to fill an area defined by a sequence of lines, you begin drawing with the command **startPath()**. Using **fillPath()**, the start point and the end point are connected and the enclosed area is filled.

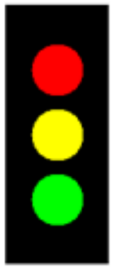
You can write **comments** by starting a code line with the # symbol.

EXERCISES

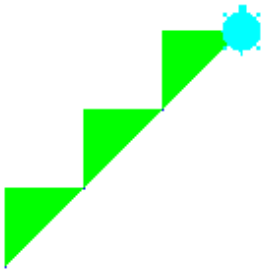
1. Draw a regular hexagon with the turtle and make each side a different color.



2. Draw a traffic light. You can draw the black rectangle with a pencil width of 80 and the circles with *dot(40)*.



3. The turtle should draw the adjacent image.



2.3 REPETITION

INTRODUCTION

Computers are particularly good at repeating the same instructions (including turtle commands) over and over again. In order to draw a square, you do not need to enter the commands *forward(100)* and *left(90)* four times in a row. It is rather sufficient to tell the turtle to simply repeat the two statements four times.

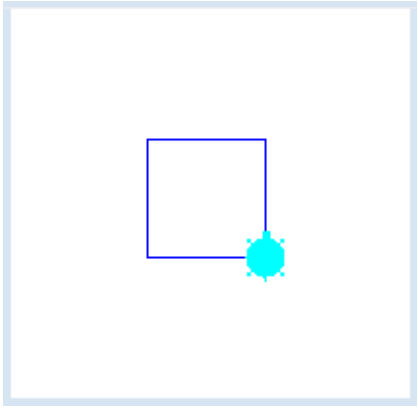
With the instruction *repeat,* you tell the turtle that some commands should repeat a designated number of times. In order for the computer to know that these commands belong together (forming a program block), they must be equally indented. Typically, we use three spaces for indentation.

PROGRAMMING CONCEPTS: *Simple repeat loop, repeat loop instead of code duplication*

REPEAT - STRUCTURE

In order to draw a square, the turtle has to move straight ahead four times and make a total of four 90° turns. If you were to write out each command separately, the program would become quite long.

With the instruction **repeat 4:** you tell the turtle to repeat the **indented lines** four times. Make sure not to forget the colon!



```
from gturtle import *

makeTurtle()
repeat 4:
    forward(100)
    left(90)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

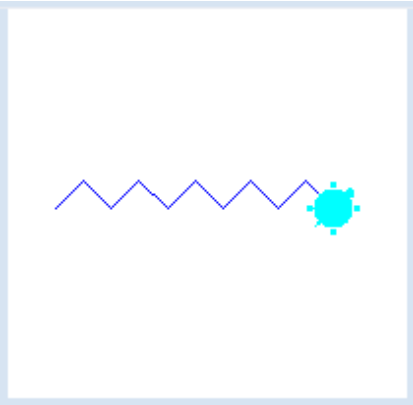
MEMO

With *repeat n:* you tell the computer that it should repeat one or more instructions *n* times before it executes further instructions. Everything that is to be repeated must be placed below repeat, and must also be indented.

```
repeat number:
    Instructions, that
    should be
    repeated
```

REPEATING SOUNDS

A typical example of a repetition is the Dah-Dih-Dah-Dih sound of fire trucks. Using the turtle you can easily create such a tone sequence, and simultaneously for fun, you can let the turtle draw a zigzag curve. You can generate a pure tone with **playTone()**, where you specify its pitch as a frequency (in Hertz) and its duration (in milliseconds).



```
from gturtle import *

makeTurtle()

setPos(-200, 0)
right(45)

repeat 5:
    playTone(392, 400)
    forward(50)
    right(90)
    playTone(523, 400)
    forward(50)
    left(90)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

With **setPos(x, y)** you can directly put the turtle into a designated position in the window without actually making a trace. The two numbers, x and y, are the coordinates relative to the zero point, which is located at the middle of the window. (The coordinate range depends on the size of the window.)

You can also specify the pitch of **playTone()** using a letter according to the musical scale, for example with c, d, etc., or in the one-line octave with c', d', etc. (or with two or three apostrophes). You have to put quotation marks around the pitch. If you want, you can also indicate a musical instrument to be used (available are: piano, guitar, harp, trumpet, xylophone, organ, violin, panflute, bird, seashore). Try it once with:

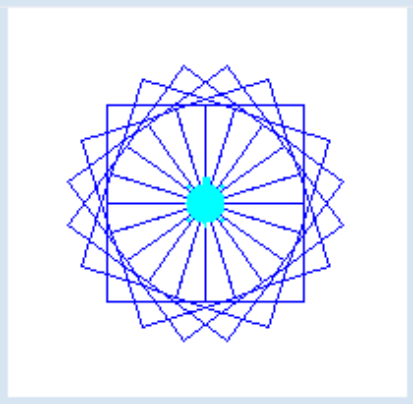
Lower Ton:	playTone("g'", 400, instrument = "trumpet")
Higher Ton:	playTone("c''", 400, instrument = "trumpet")

NESTED REPEATS

A square can be made quite easily with a four-fold repetition. Now let's draw 20 squares, with the squares slightly rotating against each other.

You first have to nest the two *repeat* statements into each other. In the **inner program block**, the turtle draws a square and then turns by 18 degrees to the right. The **outer repeat statement** repeats this 20 times. Please make sure to correctly indent the statements that should be repeated.

If you hide the turtle with **hideTurtle()**, it will finish drawing quicker.



```
from gturtle import *

makeTurtle()

# hideTurtle()
repeat 20:
    repeat 4:
        forward(80)
        left(90)
    right(18)
```

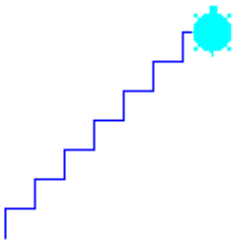
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

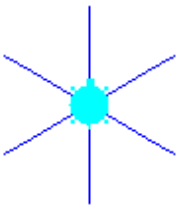
The *repeat* commands can be nested. It is very important to have the correct indentations in the statements which are to be repeated.

EXERCISES

1. Draw a staircase with seven steps.

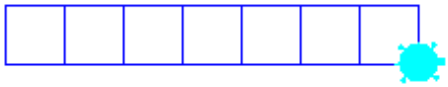


2. Draw a star using the *back()* command.

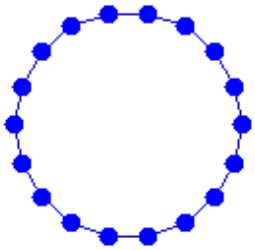


3. You can draw a "real" star with rotation angles 140° and 80°.

4. Draw the following figure using two nested *repeat* statements. The inner *repeat* block will draw a square.



5. Draw a pearl necklace.



6. Draw a bird.

2.4 FUNCTIONS

INTRODUCTION

In a larger picture, you may want to use figures such as triangles and squares repeatedly. However, the turtle itself does not know what a triangle or a square is. Therefore you have to explain to the turtle how to draw the figures each time with a complete program code. Is this possible in an easier way?

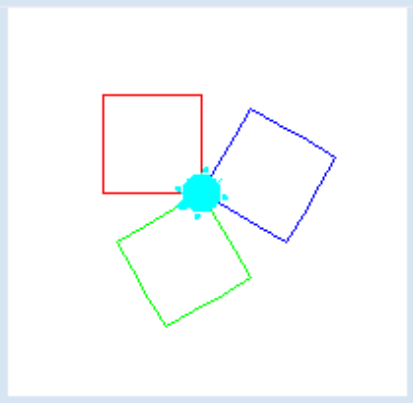
It is! You can teach the turtle new commands, for example how to draw a square or a triangle. Then you simply have to tell the turtle that it should execute such a command, namely draw a square or a triangle. In order to define a new command, you can choose any given identifier, for example *square*, and then write *def square():* After that, you then write down all of the instructions belonging to the new command. In order for the computer to know what is part of the new command, the instructions must be indented.

PROGRAMMING CONCEPTS: *Modular programming, function definition, function call*

DEFINING YOUR OWN COMMAND

In this program you will use ***def*** to define the new command **square()**. Afterwards, the turtle will know how to draw a square, however, it will not have drawn one yet.

Using the command *square()* the turtle draws a **square** at its current position with a side length of 100. In our example there is a red, a blue, and a green square.



```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        left(90)

makeTurtle()
setPenColor("red")
square()
right(120)
setPenColor("blue")
square()
right(120)
setPenColor("green")
square()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

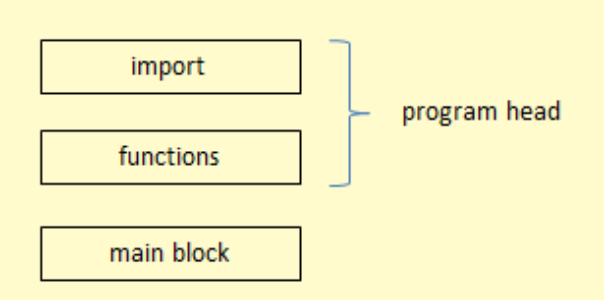
MEMO

You can define a new command using ***def identifier():*** Choose a name that reflects the activity of the command. All instructions that belong to the new command must be indented

```
def identifier():
    instuctions
```

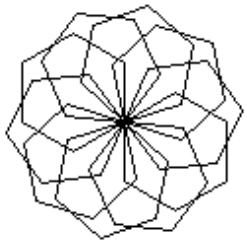
Do not forget to put brackets and the colon after the identifier! In *Python* you also call new commands *functions*. When you use the function *square()* one could also say that the function is "called".

We should get used to placing the function definitions in the program header, since they have to be defined before they are called

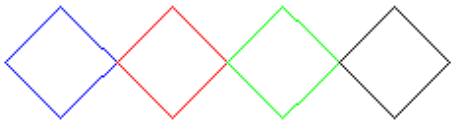


EXERCISES

1. Define a command *hexagon()* with which you can draw a hexagon, then use this command to draw the adjacent figure.



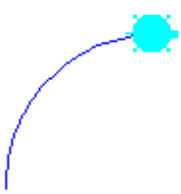
2a. Define a command that draws a square standing on one of its corners, then use this command to draw the adjacent figure.



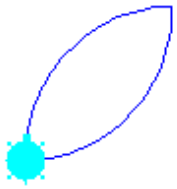
2b*. You can draw filled squares by using the commands *startPath()* and *fillPath()*.

3a. In this task, you will experience how to solve a problem step by step using functions [**more...**].

Define a function *arc()* that tells the turtle to draw an arc and rotate a total of 90 degrees to the right. You can set the maximum value of the turtle's speed with *speed(-1)*.

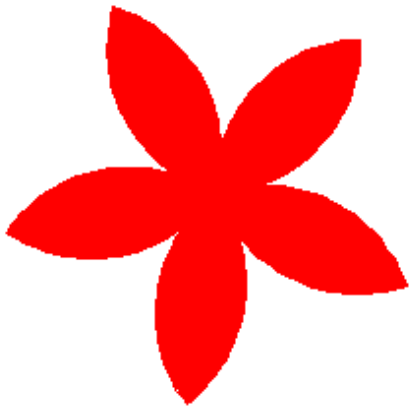


3b. Add the function *petal()* to the program, which will draw two arcs. At the end, the turtle should be back in the original starting direction.

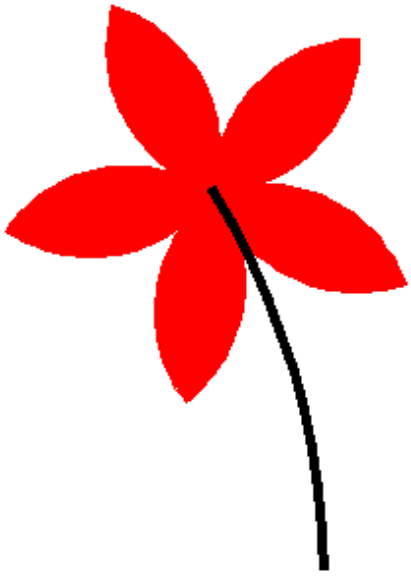


3c. Add another command to the program so that the *petal()* is drawn as a filled red leaf (without a visible border line).

3d. Extend the program with the function *flower()*, which draws a five-petalled flower. To make the turtle draw the flower even faster, use the function *hideTurtle()* to make the turtle invisible.



3e*. Add a stem to the flower.



2.5 PARAMETER

EINFÜHRUNG

Beim Befehl *forward()* gibst du in Klammern an, um welche Strecke die Turtle vorwärts gehen soll. Dieser Wert in den Klammern gibt an, wie weit vorwärts gegangen wird. Er präzisiert den Befehl und heisst ein Parameter: Hier ist es eine Zahl, die bei jeder Verwendung von *forward()* anders sein kann. Im vorhergehenden Kapitel hast du einen eigenen Befehl *square()* definiert. Im Unterschied zu *forward()* ist die Seitenlänge dieses Quadrats aber immer 100 Pixel. Dabei wäre es doch in vielen Fällen praktisch, die Seitenlänge des Quadrats anpassen zu können. Wie geht das?

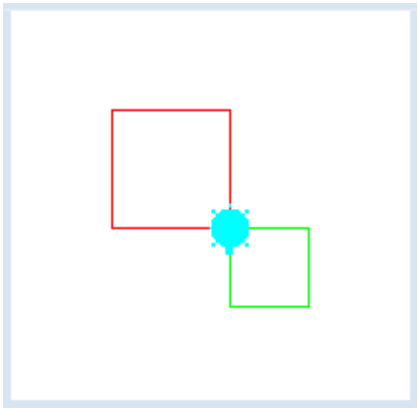
PROGRAMMIERKONZEPTE: *Parameter, Parameterübergabe*

BEFEHLE MIT PARAMETER

Auch in diesem Programm definieren wir ein Quadrat. An Stelle der leeren Parameterklammer bei der Definition der Funktion *square(),* setzen wir den Parameternamen *sidelength* ein und verwenden diesen beim Aufruf von **forward(sidelength)**.

Du kannst dadurch *square* mehrmals verwenden und bei jeder Verwendung eine Zahl für *seite* angeben.

Mit **square(80)** zeichnet die Turtle ein Quadrat mit der Seitenlänge von 80 Pixeln, mit **square(50)** eines mit der Seitenlänge von 50 Pixeln.



```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
setPenColor("red")
square(80)
left(180)
setPenColor("green")
square(50)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Parameter sind Platzhalter für Werte, die jedes Mal anders sein können. Du gibst den Parameter bei der Definition eines Befehls hinter den Befehlsnamen in einem Klammerpaar an.

```
def befehlsname(parameter):
    Anweisungen, die
    parameter verwenden
```

Der Parametername ist frei wählbar, sollte aber seine Bedeutung wiederspiegeln. Bei der

Verwendung des Befehls gibst du wieder in Klammern den Wert an, den der Parameter haben soll.

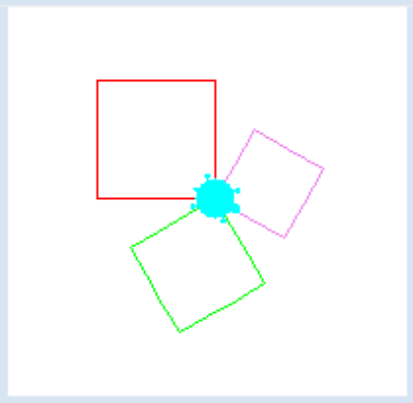
```
befehlsname(123)
```

Hier wird der Parameter im ganzen Befehl durch 123 ersetzt.

MEHRERE PARAMETER

Befehle können mehrere Parameter besitzen. Beim Quadrat kannst du zum Beispiel mit **def square(sidelength, color)** als Parameter seite und farbe wählen.

Du kannst dann *quadrat* viel flexibler verwenden.Mit **square(100, "red")** zeichnet die Turtle ein rotes Quadrat mit der Seitenlänge von 100 Pixeln, mit **square(80, "green")** ein grünes mit der Seitenlänge von 80 Pixeln.



```
from gturtle import *

def square(sidelength, color):
    setPenColor(color)
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
square(100, "red")
left(120)
square(80, "green")
left(120)
square(60, "violet")
```

MEMO

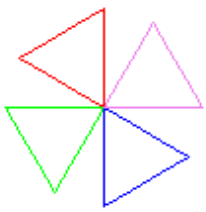
Befehle können mehrere Parameter besitzen. Diese werden in der Parameterklammer getrennt mit Komma eingegeben.

```
def befehlsname(parameter1, parameter2....):
   Anweisungen, die parameter1
   und parameter2 verwenden
```

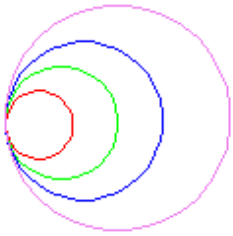
Die Reihenfolge der Parameter in der Parameterklammer bei der Definition des Befehls muss mit der Reihenfolge der Werte beim Aufruf des Befehls übereinstimmen.

AUFGABEN

1. Definiere einen Befehl *triangle(color)*, mit welchem die Turtle farbige Dreiecke zeichnen kann. Zeichne 4 Dreiecke in den Farben red, green, blue und violet



2. Definiere einen Befehl *colorCircle(radius, color)*, mit welchem die Turtle einen farbigen Kreis zeichnet. Du kannst dabei den Befehl *rightArc(radius, angle)* verwenden. Zeichne die nebenstehende Figur.



3. Das folgende Programm zeichnet leider 3 gleich grosse Fünfecke, aber nicht wie gewünscht verschieden grosse. Warum nicht? Korrigiere es.

```
from gturtle import *

def pentagon(sidelength, color):
   setPenColor(color)
   repeat 5:
      forward(90)
      left(72)

makeTurtle()
pentagon(100, "red")
left(120)
pentagon(80, "green")
left(120)
pentagon(60, "violet")
```

4. Du sagst der Turtle mit dem Befehl segment(), sich um eine bestimmte Strecke s vorwärts zu bewegen und sich um einen bestimmten Winkel w nach rechts zu drehen:

```
def segment(s, w):
   forward(s)
   right(w)
```

Schreibe ein Programm, das diesen Befehl 92 mal mit s = 300 und w = 151 ausführt. Mit *setPos(x, y)* kannst du die Turtle zu Beginn geeignet im Fenster positionieren.

5*. Die Turtle soll zwei, drei oder vier segment-Bewegungen ausführen. Schau dir die schönen Grafiken in folgenden Fällen an:

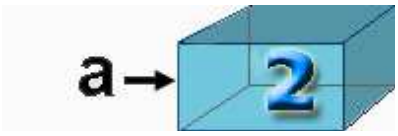
Anzahl Segmente	Werte	Anzahl Wiederholungen
2	forward(77) right(140.86) forward(310) right(112)	37
3	forward(15.4) right(140.86) forward(62) right(112) forwad(57.2) right(130)	46
4	forward(31) right(141) forward(112) right(87.19) forward(115.2) right(130) forward(186) right(121.43)	68

2.6 VARIABLES

INTRODUCTION

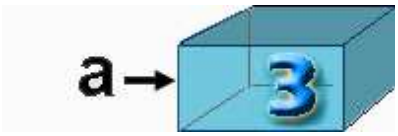
In the previous chapter, you drew squares with side lengths that were firmly implemented in the program. There may also be times when you want to enter the side length with an input dialog. In order to do this, the program needs to store the entered number as a **variable**. You can see the variable as a container, the content of which you can access with a name. So, in short, a variable has **a name and a value**. You can freely choose the name of a variable, but they cannot be keywords or names with special characters. Moreover the name cannot start with a number.

With the notation *a = 2* you create the container that you can access with the name *a* and put the number 2 inside. In the future we will say that you are **defining** a variable *a* and **assigning** a value to it.



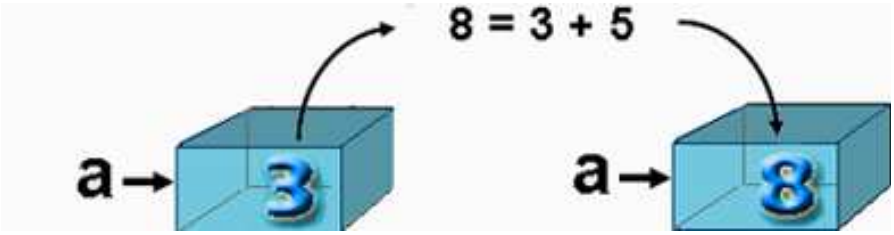
a = 2 : variable definition (assignment)

You can only put **one** object into the container. Later in your program, when you want to store another number 3 under the name *a*, write *a = 3* [**more...**].



a = 3 : new assignment

So then what happens when you write **a = a + 5**? You take the number that is currently in the container, accessed with the name *a,* and therefore the number 3 is added to the number 5. The result adds up to 8 and it is again saved under the name *a*.

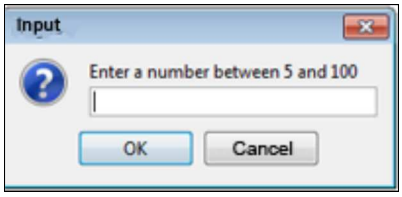


Therefore, the equal sign does not mean the same thing in computer programming as it does in mathematics. It does not define an equation, but rather a variable definition or an assignment [**more...**].

PROGRAMMING CONCEPTS: *Variable definition, assignment*

READING AND MODIFYING VARIABLE VALUES

You can assign a value between 10 and 100 to the **variable x** with the help of the dialog box. You **change** this value in the following looping structure, which results in drawing a spiral.




```
from gturtle import *

makeTurtle()
```

```
x = inputInt("Enter a number between 5 and 100")
repeat 10:
    forward(x)
    left(120)
    x = x + 20
```

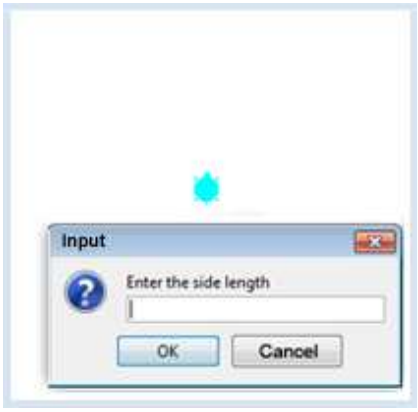
MEMO

With *variables* you can store values that you are able to read and change in the course of the program. Every variable has a name and a value. You can define a variable and assign a value to it using an equal sign [**more...**].

DISTINGUISHING VARIABLES AND PARAMETERS

You should be aware of the differences between a variable and a parameter. Parameters transport data into a function and are only valid within that function, whereas variables are possible anywhere. When calling a function, you give each of its parameters values that can be used as variables within the function's scope.

To make the difference clear, use the parameter *sidelength* in the function *square()* in your program. When you input a number with **inputInt()**, it stores it as the *variable s*. When you call **square(),** you then pass the variable value of *s* on to the parameter *sidelength*.




```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
s = inputInt("Enter the side length")
square(s)
```

MEMO

You have to distinguish between the variable *s* and the parameter *sidelength*. In the definition of a function parameters are placeholders and can be regarded as variables that are only known inside of the function each time it is called. If you call the function with a variable, the variable's value is used in the function. Thus, *square(length)* draws a square with a side length of *length* [**more...**].

THE SAME NAME FOR DIFFERENT THINGS

As you already know, parameters and variables should be named after what they relate to, but they can be chosen arbitrarily. Because of this, it is common to choose the same name for **parameters** and **variables**. No naming conflicts arise. However, you must remember the distinction in order to understand the program.

```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
sidelength = inputInt("Enter the side length")
square(sidelength)
```

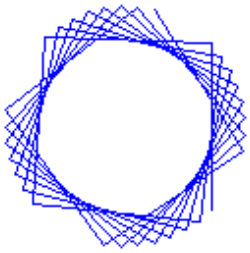
MEMO

Although you can use the same name for a specific **parameter** and **variable,** you should be able to conceptualize them separately.

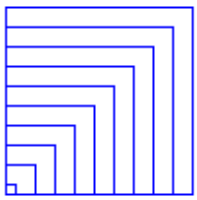
EXERCISES

1. After entering the number of corners into the dialog box, the turtle should draw a regular *n*-gon. For example, when you input the number 8, an 8-gon (octagon) should be drawn. The program should calculate the appropriate rotation angle. Put yourself in the position of the turtle and think how far you have to rotate yourself in order to draw the next side. Remember how we drew an equilateral triangle. .

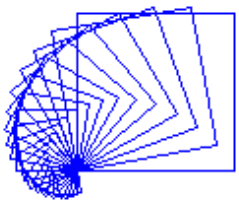
2. After entering an angle in the dialog box, the turtle draws 30 lines, each with a side length of 100, and after each of which it rotates left by the given angle. Experiment with different angles and draw some cool pictures. You can speed up the drawing by with *hideTurtle()*.



3. Tell the turtle to draw 10 squares. First define a command *square* with the parameter *sidelength*. The side length of the first square is 8, and each following square has a side length increasing by 10



4. Enter the side length of the largest square into the dialog box. The turtle will then draw 20 squares. After each previous square, the side length should be smaller by a **factor** of 0.9 and the turtle should rotate 10° to the left



2.7 SELECTION

INTRODUCTION

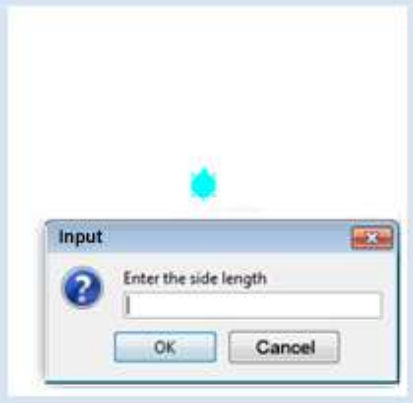
What you do in your daily life often depends on certain conditions. So let's say you decide on how you will get to school today depending on the weather. You say: "*In case it rains today, I'll take the tram, otherwise I'll ll ride my bike*". Similar to this, flow of a program can also depend on certain conditions. Among the basic structures of any programming language are such program branches that depend on specific conditions. The instructions after *if* are only executed when the condition is true, otherwise the statements after *else* are executed.

PROGRAMMING CONCEPTS: *Condition, program branching, selection, if-else structure*

REVIEWING INPUTS

After you **enter the side length** in the dialog box, the square will only be drawn if it fits the window entirely.

We now examine the value of *s*. **If *s* is less than 300** a square with the side length *s* is drawn, **otherwise** a message appears in the lower part of the Tigerjython window. In a programming language, this test is done using the *if* statement.




```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        right(90)

makeTurtle()
s = inputInt("Enter the side length")
if s < 300:
    square(s)
else:
    print "The side length is too big"
```

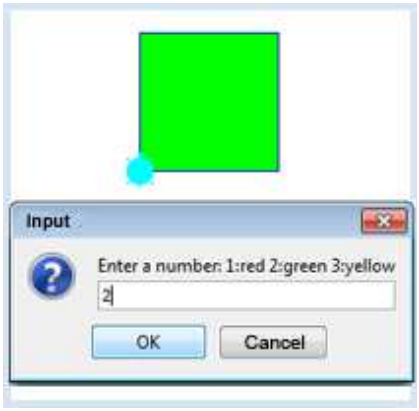
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The instructions after ***if*** are only executed if the condition is true, otherwise the statements after ***else*** are executed. You can also leave out the *else* block. Try it!

Please be aware of the colons after the *if* condition and after *else*, as well as a correct indentation of both program blocks.

MULTIPLE SELECTION



Now we would like to draw colored squares. You can enter the desired color by putting a **number** into the dialog box. In an **if structure**, the number is checked and then the appropriate fill color is set. We first test if the value is 1, then if it is 2 (with **elif**), and finally **3**. For any other number entered, we use **else** to set the color to black.

With the command **fill(10, 10)** the closed area around the given point is filled with the specified fill color. Since after drawing the square, the turtle is back in the center of the window (0, 0), by using (10, 10) we select a point that is definitely inside of the square.

```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        right(90)

makeTurtle()
n = inputInt("Enter a number: 1:red 2:green 3:yellow")
if n == 1:
    setFillColor("red")
elif n == 2:
    setFillColor("green")
elif n == 3:
    setFillColor("yellow")
else:
    setFillColor("black")

square()
fill(10, 10)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Several conditions can be checked consecutively. In a case where the condition at *if* is not fulfilled, the condition at *elif* is checked. *elif* is an abbreviation of *else if*. In a case where none of the *elif* conditions are fulfilled, any statements after *else* are executed.
It is very important to remember that in Python, a double equal sign is used in the test for equality. It may take time getting used to it, but it is necessary because the single equal sign is used for assignments.

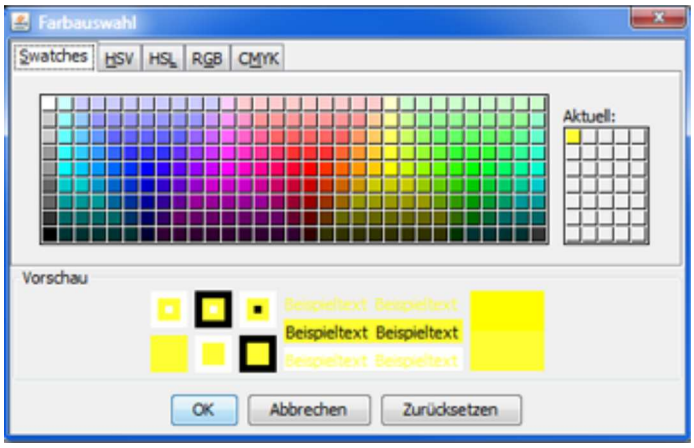
Please be aware of the notations used for comparison operators: >, >= , < , <= , == , != .

With the command *fill(x, y)* you can fill closed figures with the fill color. However, the point (x, y) must be located inside of the figure.

COLOR CHOICE, BOOLEAN VARIABLES

In order to fill a figure afterwards using the *fill()* statement, its interior of the figure cannot already be occupied by another figure. You already know the **startPath()**/**fillPath()**combination with which you can correctly fill new figures that lay on top of existing figures.

In this program you call **askColor()**, which brings up a nice dialog box with which you can choose the color of the star.



The star that you draw uses the function **star()**, which in addition to the size of the star also has a parameter *filled* whose value can be *true* or *false* and determines whether the star should be filled or not.

```
from gturtle import *

makeTurtle()

def star(size, filled):
    if filled:
        startPath()
    repeat 9:
        forward(size)
        left(175)
        forward(size)
        left(225)
    if filled:
        fillPath()

clear("black")
repeat 5:
    color = askColor("Color selection", "yellow")
    if color == None:
        break
    setPenColor(color)
    setFillColor(color)
    setRandomPos(400, 400)
    back(100)
    star(100, True)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The function **askColor()** has parameters for the text in the title bar and the color that is selected as the default value. When you click the OK button the function returns the selected color and when you click the cancel button instead, the function returns the special value *none*. You can test this value with an *if* statement and you can abort the repeat loop with *break*.

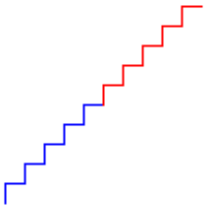
A variable or a parameter which can take the values *true* or *false* is called a boolean variable or a boolean parameter [**more...**]. You can directly test its value, for instance with **if filled:** It is thus not necessary (and not very elegant) to write *if filled == True.*

EXERCISES

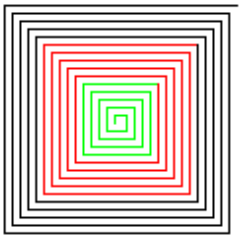
1. In the dialog box you ask the user how big the side length of a square should be. If it is less than 50, a red square is drawn with this side length. Otherwise a green square is drawn.

2. Make the turtle draw a staircase with 10 steps using *repeat 10*. Make the first 5 levels blue and the rest of the levels red (Figure a).

(a)



(b)



Tell the turtle to draw a spiral, first using green, then red, and finally black (Figure b).

2.8 WHILE LOOPS

INTRODUCTION

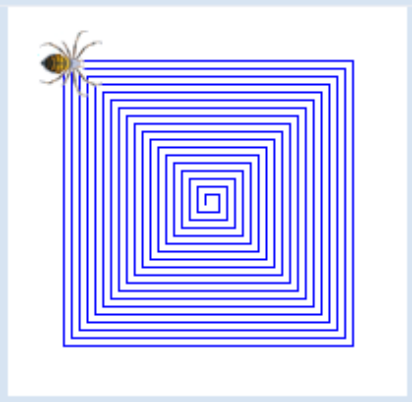
You have already gotten to know the command *repeat*, with which you can repeat a program block several times. However, it is important to know that you can only use *repeat* this way in TigerJython and not in other Python flavours. On the other hand, you can use the *while* structure everywhere.
The *while* loop is initiated with the keyword *while*, followed by a looping condition. The instructions in the loop block are repeated as long as the condition is fulfilled. The program then continues on with the next statement listed after the loop block.

PROGRAMMING CONCEPTS: *Iteration, while structure, combined conditions, loop termination*

SPIDER WEB

The turtle should draw a rectangular spiral with the help of a *while* loop. We will use a variable *a*, with an **initial value 5**which is then **increased by 2** with each iteration of the loop. As long as the **condition a < 200** is true, the statements in the loop block will be executed.

To make it a bit more fun, you can switch out the turtle icon for a spider.



```
from gturtle import *

makeTurtle("sprites/spider.png")

a = 5
while a < 200:
    forward(a)
    right(90)
    a = a + 2
```

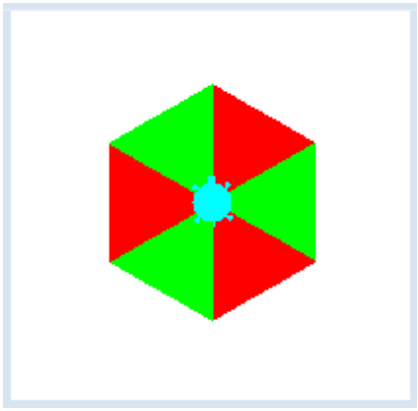
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

A *while* loop is used to repeat a program block. In order for the program block to be executed, the condition must be true. Because of this, one might also call it a *"running condition*". If the change in value is missing in the loop block, the running condition always stays true and the program remains endlessly "hanging" in the loop.

In our learning environment, you can cancel the hanging program with the stop button or by closing the turtle window. In general, infinite loops without a the option to cancel are dangerous, and in an extreme case you will have to reboot your computer.

COMBINING CONDITIONS WITH OR



The turtle should draw the adjacent figure using the **while loop.** As you can see, it is drawn with alternating red and green triangles.

You can use the following trick to change the colors: Test the loop variable to see whether it is 0, 2 or 4 and then choose the pen color red.

Using the command **fillToPoint(0. 0)** you can fill a figure with color while drawing. In this case, it is as though a rubber band was attached to the point (0, 0), the other end of which the turtle drags along. All points the rubber band reaches along the way are colored consecutively.

```
from gturtle import *

def triangle():
    repeat 3:
        forward(100)
        right(120)

makeTurtle()
i = 0
while i < 6:
    if i == 0 or i == 2 or i == 4:
         setPenColor("red")
    else:
         setPenColor("green")

    fillToPoint(0, 0)
    triangle()
    right(60)
    i = i + 1
```

MEMO

You must pay attention to the correct indentation for each loop block when using several program structures. As you can see, you can combine two or more conditions using *or*. A condition linked this way is true if either of the conditions are satisfied, and it is also true if both conditions are met. Using the command *fillToPoint(x, y)* you can fill figures with the pen color while drawing, as opposed to the command *fill(),* with which you can fill already drawn closed figures.

COMBINING CONDITIONS WITH AND

The turtle should draw 10 connected houses using a *while* loop. The houses are numbered from 1 to 10. The houses with the numbers 4-7 are large, and all of the other houses are small. In the *while* loop, the house number *nr* is used to determine the size of the houses. The houses are large if *nr* is greater than 3 and less than 8.

We use the command **fillToHorizontal(0)** to add color. As a result, the area between the drawn figure and the horizontal line y = 0 is filled consecutively.

```
from gturtle import *

makeTurtle()
setPos(-200, 30)
right(30)
fillToHorizontal(0)
setPenColor("sienna")

nr = 1
while nr <= 10:
    if nr > 3 and nr < 8:
        forward(60)
        right(120)
        forward(60)
        left(120)
    else:
        forward(30)
        right(120)
        forward(30)
        left(120)

    nr += 1
```

MEMO

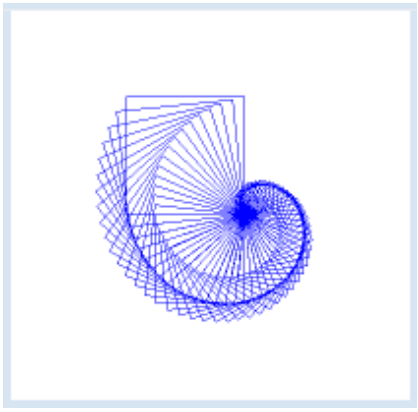
You can link two conditions with **and**. Such a linked statement is only true if both conditions are met. Using the command **fillToHorizontal(y)** you can fill figures with the pen color while drawing. This way, the area between the drawn figure and the horizontal line at *y* is filled.

nr += 1 means that *nr* is increased by 1. It is just an abbreviation for the assignment *nr = nr + 1*.

EXITING LOOPS WITH BREAK

A loop whose condition is always true will loop forever. However, you can force a loop to exit at any time using the keyword **break**.

Your program will draw rotated squares with increasing side lengths until the side length is 120.



```
from gturtle import *

def square(sidelength):
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
hideTurtle()

i = 0
while 1 == 1:
    if i > 120:
```

```
        break
    square(i)
    right(6)
    i += 2
print "i =", i
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Instead of using **while 1 == 1:** you can use *while True:*, since *True* is always true. (On the other hand, *False* is always false.)

The loop is run in increments of two. Instead of using i = i + 2 you should use the abbreviated notion **i += 2** (*i* is incremented by 2).

With the command **print**-you can write something into the TigerJython console at the bottom of the editor. With text you should use quotation marks and numbers should be separated by a comma. A space will automatically be inserted between the text and the number. Do you understand why the output is *i* = 122?

The keyword ***continue*** is rarely used. It skips the remaining part of the body of the loop.

INPUT VALIDATION

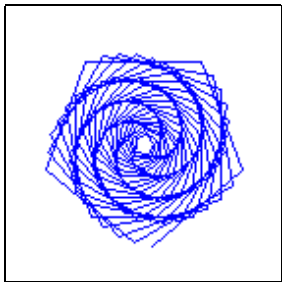
If you ask the user to enter a number restricted to a certain range, you cannot trust him that he adheres to your restriction. A "robust" program checks the input and intercepts an incorrect entry with a feedback. This input validation is most easily performed in a while loop which is repeated until the input value is accepted. In your program the user enters the number 1, 2, or 3 to select one the colors red, green, or yellow of the filled circle.

```
from gturtle import *

makeTurtle()

n = 0
while n < 1 or n > 3:
    n = inputInt("Enter 1, 2 or 3")
if n == 1:
    setPenColor("red")
elif n == 2:
    setPenColor("green")
else:
    setPenColor("yellow")
dot(200)
```

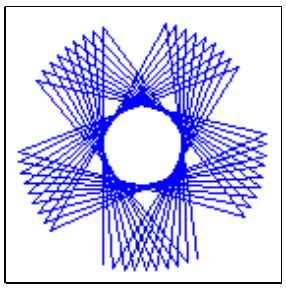
EXERCISES

1. The turtle moves forward on a line with length 5 and turns 70° to the right. Then it increases the line length by 0.5 and repeats these steps as long as the line length is smaller than 150.

   Try it also with the rotation angle of 89 °!

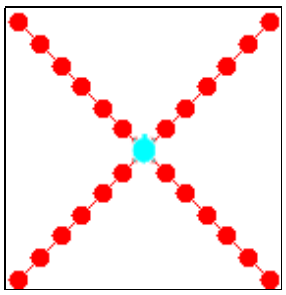


2. As you probably noticed, the rotation angle at a tip of the 5 edged star is 144°. Change this rotation angle by just a little bit, for example to 143°, and increase the number of repetitions. You will then get a new figure.



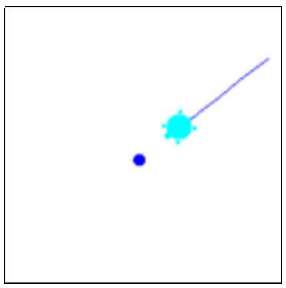
3. The turtle draws a diagonal pattern with filled red circles. All circles are located in the Turtle window, which means that the distance from the center is less than 400.

   Use the command dot(25) for creating the circles.



4. The turtle is located at the position (250, 200). With steps of length 10, the turtle moves on a straight line to the home position until the distance is less than 1.

   Use the commands *towards()* and *heading(degrees)* (see **documentation**).



5*. In order to stop the turtle more precisely at home, you decrease the distance by a factor 10 to 100. Unfortunately it may happen that the turtle does not stop anymore. Can you explain this behavior?

2.10 EVENT CONTROL

INTRODUCTION

So far we have only seen a programs with a single strand of events, where one statement after another is executed, with possible ramifications and repetitions. However, when you click a mouse button, for instance, while your program is being executed, you cannot be sure where your program is located at the time. In order to capture the clicks in the program we have to introduce a **new programming concept** called **event control**. The principle is as follows:

Define a function with any name, for example *onMouseHit()* that is never explicitly called in the program. Then ask **that your computer** calls this function whenever the mouse button is clicked. So, what you are telling the program is: *Whenever the mouse button is clicked, execute onMouseHit()*.

PROGRAMMING CONCEPTS: *Event-driven programming, mouse event*

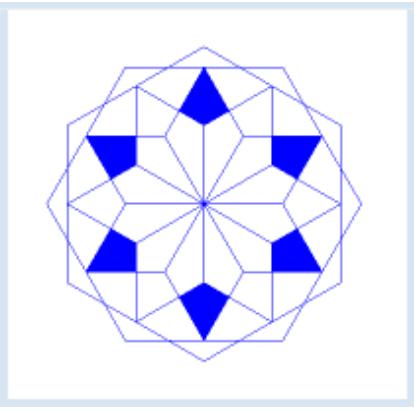
MOUSE EVENTS

It is very easy to implement the new concept in Python. In the first event-driven program, the turtle should draw a fun figure in the main part. After this, you can decorate it by coloring certain areas with a mouse click.

Write the function **onMouseHit(x, y)**, which delivers the x- and y-coordinate of the mouse click, and then give it a flood fill by using **fill(x, y)** (the filling of a closed region).

Most importantly you have to **tell the system** that it should call the function *onMouseHit()* whenever the mouse button is pressed. In order to do this you can use the parameter **mouseHit** when you call *makeTurtle()* and assign it the name of your function.

Use *hideTurtle()* so that the drawing is created faster.



```
from gturtle import *

def onMouseHit(x, y):
    fill(x, y)

makeTurtle(mouseHit = onMouseHit)
hideTurtle()
addStatusBar(30)
setStatusText("Click to fill a region!")

repeat 12:
    repeat 6:
        forward(80)
        right(60)
    left(30)
```

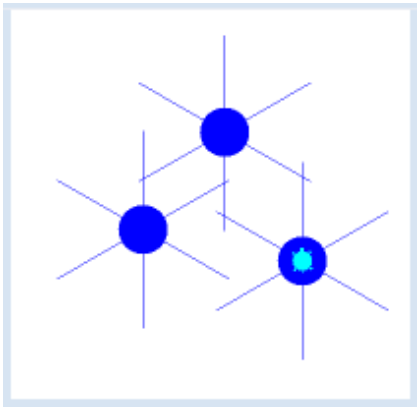
MEMO

Technically, the concept of event-driven programming is implemented by writing a function to be called whenever the event occurs. You inform the system which function this is by passing the name of your function to *makeTurtle()*. Here you are using the notation *parameter_name = parameter_value*.
You can customize your preferences for the fill color with *setFillColor()*.

You can write important information for the user in a status bar below the turtle window by using **addStatusBar(n)**. The number *n* states the line height of the text bar (in pixels).

DRAWING WITH A MOUSE CLICK

The turtle should draw a star with rays at the position of the mouse click. For this you write the function **onMouseHit(x, y)** where you instruct the turtle how to draw the star. In order for *onMouseHit()* to be called when the mouse is clicked you pass the parameter name **mouseHit** in *makeTurtle()* the function name *onMouseHit.*



```
from gturtle import *

def onMouseHit(x, y):
    setPos(x, y)
    repeat 6:
        dot(40)
        forward(60)
        back(60)
        right(60)

makeTurtle(mouseHit = onMouseHit)
speed(-1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

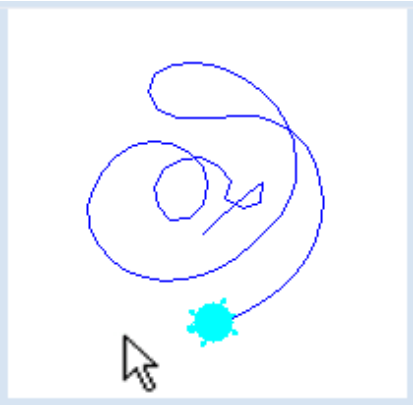
The program has a flaw: If you click again while the turtle is still drawing a star, it will not finish that star but will immediately begin drawing the new star. However, it also continues to execute the commands of the old star, and so the new star is drawn incorrectly.

This wrong behavior is apparently due to the fact that every time you click, the function *onMouseHit()* it is called and executed, even if the previous execution is not done yet. In order to prevent this from happening, you can use the parameter named *mouseHitX* instead of *mouseHit*.

TURTLE CHASING THE MOUSE

Now you want the turtle to follow the mouse everywhere it goes. You cannot use the actual mouse click, but instead you should consider the **movement of the mouse** as an event. *makeTurtle()* knows the parameter *mouseMoved* to which you can pass a function that is called at every relocation of the mouse.

The function **onMouseMoved(x, y)** receives the current mouse coordinates x and y.



```
from gturtle import *

def onMouseMoved(x, y):
    setHeading(towards(x, y))
    forward(10)

makeTurtle(mouseMoved = onMouseMoved)
speed(-1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Besides *mouseHit* and *mouseHitX* there are other parameters of *makeTurtle()* at your disposal with which you can detect mouse events. Instead of x, y they use *event*, from which you can determine the coordinates of the mouse event.

mousePressed	Mouse button is pressed
mouseReleased	Mouse button is released
mouseClicked	Mouse button is pressed and released
mouseDragged	Mouse is moved while the button is pressed
mouseMoved	Mouse is moved
mouseEntered	Mouse enters the turtle window
mouseExited	Mouse exits the turtle window

You can also use multiple parameters simultaneously, for example the two functions *onMousePressed()* and *onMouseDragged()*:

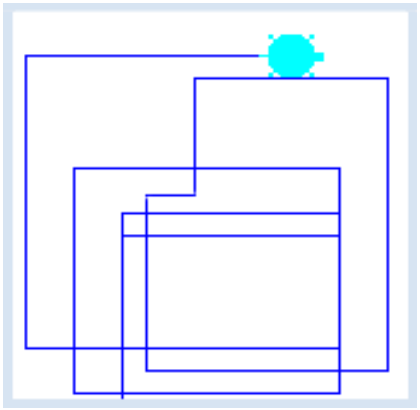
```
makeTurtle(mousePressed = onMousePressed, mouseDragged = onMouseDragged)
```

You can find out which mouse button was pressed with *isLeftMouseButton()* or *isRightMouseButton()*.

There is an important difference between these events and *mouseHit*: the movement of the turtle is not visible during the execution of the function. Therefore, you should either set the turtle on high speed with *speed(-1),* hide it with *hideTurtle()*, or execute the code for its movement in the main part of the program.

KEY EVENTS

Each time a keyboard key is hit, an event is "fired". To handle it, you register a callback function in *makeTurtle* by using the named parameter *keyPressed*. The callback receives an integer code that identifies the key you pressed. (You may find out the key codes by performing some simple tests.) In your program the turtles moves repeatedly 10 steps in forward direction. By hitting the cursor keys you can change its orientation in the four cardinal directions. To prevent the turtle to leave the playground, the wrap mode is enabled.

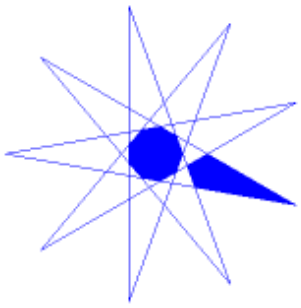


```
from gturtle import *

LEFT = 37
RIGHT = 39
UP = 38
DOWN = 40

def onKeyPressed(key):
   if key == LEFT:
      setHeading(-90)
   elif key == RIGHT:
      setHeading(90)
   elif key == UP:
      setHeading(0)
   elif key == DOWN:
      setHeading(180)

makeTurtle(keyPressed = onKeyPressed)
wrap()
while True:
    forward(10)
```

Highlight program code (Ctrl+C kopieren, Ctrl+V einfügen)

EXERCISES

1. Draw the adjacent star with a looping structure and fill it with mouse clicks so that it suits your taste.



2. You can use the turtle to create a program where you can draw in freehand. To do this, lower the pen using the *press* event and move it using the *drag* event.

3. By pressing the left mouse button you draw any figure you would like. You can then color an area by clicking the right mouse button.

EXTRA MATERIAL

YOUR PERSONAL MOUSE IMAGE

You are able to change the image of the mouse cursor to whatever you would like, thus giving your program a special look. To do this, use the command *setCursor()* and give it one of the values from the table below. You can even use your own image if you use **setCustomCursor()**and pass it the path to your image. A standard mouse icon is 32x32 pixels in size and has a transparent background. It should be saved in gif or png format. Both *pencil.gif* and *cutemouse.gif* are already available in the distribution of *TigerJython* in the folder *sprites.*

You can now decorate the tracking program shown above with *cuteturtle* or your own mouse figure. Make sure that the turtle always moves to the mouse by using **moveTo()**.

```
from gturtle import *

def onMouseMoved(x, y):
      moveTo(x, y)

makeTurtle(mouseMoved = onMouseMoved)
setCustomCursor("sprites/cutemouse.gif")
speed(-1)
```

MEMO

By using **speed(-1)** you prevent the turtle from animating so that drawing with *moveTo()* gets faster. Possible parameters of *setCursor()*:

Parameter	**Icon**
Cursor.DEFAULT_CURSOR	Default icon
Cursor.CROSSHAIR_CURSOR	Crosshair
Cursor.MOVE_CURSOR	Moving cursor (cross arrows)
Cursor.TEXT_CURSOR	Text cursor (vertical line)
Cursor.WAIT_CURSOR	Waiting cursor

The *sprites* directory in the path indication of *setCustomCursor()* is in the same directory as your program.

2.10 EVENT CONTROL

INTRODUCTION

So far we have only seen a programs with a single strand of events, where one statement after another is executed, with possible ramifications and repetitions. However, when you click a mouse button, for instance, while your program is being executed, you cannot be sure where your program is located at the time. In order to capture the clicks in the program we have to introduce a **new programming concept** called **event control**. The principle is as follows:

Define a function with any name, for example *onMouseHit()* that is never explicitly called in the program. Then ask **that your computer** calls this function whenever the mouse button is clicked. So, what you are telling the program is: *Whenever the mouse button is clicked, execute onMouseHit()*.

PROGRAMMING CONCEPTS: *Event-driven programming, mouse event*

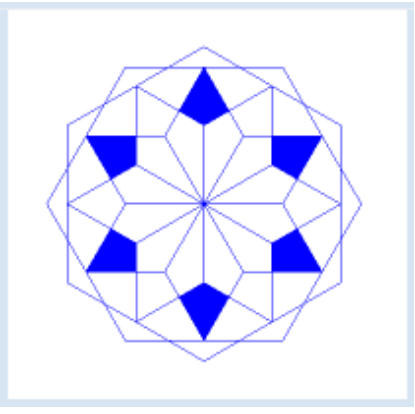
MOUSE EVENTS

It is very easy to implement the new concept in Python. In the first event-driven program, the turtle should draw a fun figure in the main part. After this, you can decorate it by coloring certain areas with a mouse click.

Write the function **onMouseHit(x, y)**, which delivers the x- and y-coordinate of the mouse click, and then give it a flood fill by using **fill(x, y)** (the filling of a closed region).

Most importantly you have to **tell the system** that it should call the function *onMouseHit()* whenever the mouse button is pressed. In order to do this you can use the parameter **mouseHit** when you call *makeTurtle()* and assign it the name of your function.

Use *hideTurtle()* so that the drawing is created faster.



```
from gturtle import *

def onMouseHit(x, y):
    fill(x, y)

makeTurtle(mouseHit = onMouseHit)
hideTurtle()
addStatusBar(30)
setStatusText("Click to fill a region!")

repeat 12:
    repeat 6:
        forward(80)
        right(60)
    left(30)
```

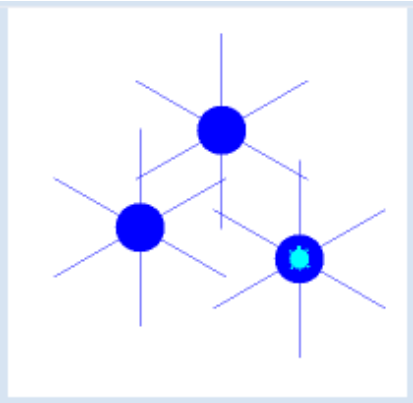
MEMO

Technically, the concept of event-driven programming is implemented by writing a function to be called whenever the event occurs. You inform the system which function this is by passing the name of your function to *makeTurtle()*. Here you are using the notation *parameter_name = parameter_value*.
You can customize your preferences for the fill color with *setFillColor()*.

You can write important information for the user in a status bar below the turtle window by using **addStatusBar(n)**. The number *n* states the line height of the text bar (in pixels).

DRAWING WITH A MOUSE CLICK

The turtle should draw a star with rays at the position of the mouse click. For this you write the function **onMouseHit(x, y)** where you instruct the turtle how to draw the star. In order for *onMouseHit()* to be called when the mouse is clicked you pass the parameter name **mouseHit** in *makeTurtle()* the function name *onMouseHit.*



```
from gturtle import *

def onMouseHit(x, y):
    setPos(x, y)
    repeat 6:
        dot(40)
        forward(60)
        back(60)
        right(60)

makeTurtle(mouseHit = onMouseHit)
speed(-1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

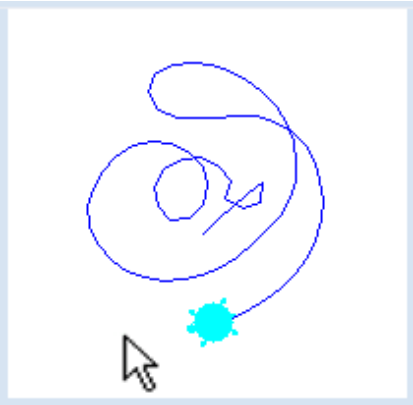
The program has a flaw: If you click again while the turtle is still drawing a star, it will not finish that star but will immediately begin drawing the new star. However, it also continues to execute the commands of the old star, and so the new star is drawn incorrectly.

This wrong behavior is apparently due to the fact that every time you click, the function *onMouseHit()* it is called and executed, even if the previous execution is not done yet. In order to prevent this from happening, you can use the parameter named *mouseHitX* instead of *mouseHit*.

TURTLE CHASING THE MOUSE

Now you want the turtle to follow the mouse everywhere it goes. You cannot use the actual mouse click, but instead you should consider the **movement of the mouse** as an event. *makeTurtle()* knows the parameter *mouseMoved* to which you can pass a function that is called at every relocation of the mouse.

The function **onMouseMoved(x, y)** receives the current mouse coordinates x and y.



```
from gturtle import *

def onMouseMoved(x, y):
    setHeading(towards(x, y))
    forward(10)

makeTurtle(mouseMoved = onMouseMoved)
speed(-1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Besides *mouseHit* and *mouseHitX* there are other parameters of *makeTurtle()* at your disposal with which you can detect mouse events. Instead of x, y they use *event*, from which you can determine the coordinates of the mouse event.

mousePressed	Mouse button is pressed
mouseReleased	Mouse button is released
mouseClicked	Mouse button is pressed and released
mouseDragged	Mouse is moved while the button is pressed
mouseMoved	Mouse is moved
mouseEntered	Mouse enters the turtle window
mouseExited	Mouse exits the turtle window

You can also use multiple parameters simultaneously, for example the two functions *onMousePressed()* and *onMouseDragged()*:

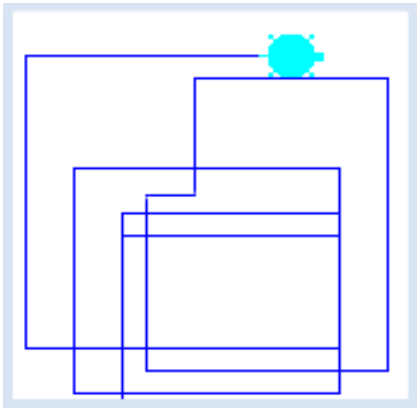
```
makeTurtle(mousePressed = onMousePressed, mouseDragged = onMouseDragged)
```

You can find out which mouse button was pressed with *isLeftMouseButton()* or *isRightMouseButton()*.

There is an important difference between these events and *mouseHit*: the movement of the turtle is not visible during the execution of the function. Therefore, you should either set the turtle on high speed with *speed(-1),* hide it with *hideTurtle()*, or execute the code for its movement in the main part of the program.

KEY EVENTS

Each time a keyboard key is hit, an event is "fired". To handle it, you register a callback function in *makeTurtle* by using the named parameter *keyPressed*. The callback receives an integer code that identifies the key you pressed. (You may find out the key codes by performing some simple tests.) In your program the turtles moves repeatedly 10 steps in forward direction. By hitting the cursor keys you can change its orientation in the four cardinal directions. To prevent the turtle to leave the playground, the wrap mode is enabled.

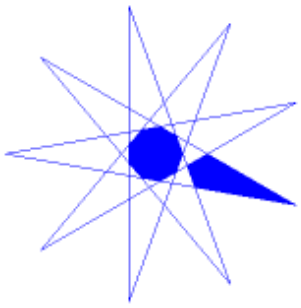


```
from gturtle import *

LEFT = 37
RIGHT = 39
UP = 38
DOWN = 40

def onKeyPressed(key):
   if key == LEFT:
      setHeading(-90)
   elif key == RIGHT:
      setHeading(90)
   elif key == UP:
      setHeading(0)
   elif key == DOWN:
      setHeading(180)

makeTurtle(keyPressed = onKeyPressed)
wrap()
while True:
    forward(10)
```

Highlight program code (Ctrl+C kopieren, Ctrl+V einfügen)

EXERCISES

1. Draw the adjacent star with a looping structure and fill it with mouse clicks so that it suits your taste.



2. You can use the turtle to create a program where you can draw in freehand. To do this, lower the pen using the *press* event and move it using the *drag* event.

3. By pressing the left mouse button you draw any figure you would like. You can then color an area by clicking the right mouse button.

EXTRA MATERIAL

YOUR PERSONAL MOUSE IMAGE

You are able to change the image of the mouse cursor to whatever you would like, thus giving your program a special look. To do this, use the command *setCursor()* and give it one of the values from the table below. You can even use your own image if you use **setCustomCursor()**and pass it the path to your image. A standard mouse icon is 32x32 pixels in size and has a transparent background. It should be saved in gif or png format. Both *pencil.gif* and *cutemouse.gif* are already available in the distribution of *TigerJython* in the folder *sprites.*

You can now decorate the tracking program shown above with *cuteturtle* or your own mouse figure. Make sure that the turtle always moves to the mouse by using **moveTo()**.

```
from gturtle import *

def onMouseMoved(x, y):
      moveTo(x, y)

makeTurtle(mouseMoved = onMouseMoved)
setCustomCursor("sprites/cutemouse.gif")
speed(-1)
```

MEMO

By using **speed(-1)** you prevent the turtle from animating so that drawing with *moveTo()* gets faster. Possible parameters of *setCursor()*:

Parameter	**Icon**
Cursor.DEFAULT_CURSOR	Default icon
Cursor.CROSSHAIR_CURSOR	Crosshair
Cursor.MOVE_CURSOR	Moving cursor (cross arrows)
Cursor.TEXT_CURSOR	Text cursor (vertical line)
Cursor.WAIT_CURSOR	Waiting cursor

The *sprites* directory in the path indication of *setCustomCursor()* is in the same directory as your program.

2.11 TURTLE OBJECTS

INTRODUCTION

In nature, a turtle is an individual with its own specific identity. In an exhibition at the zoo you could give each turtle its own name, for example Pepe or Maya. However, turtles also have things in common: they are reptiles belonging to the animal class of tortoises. These notions of classes and individuals have been so successful that they were introduced into computer science as a fundamental concept, called **object-oriented programming** (**OOP**). It will be easy for you to learn the basic principles of OOP using turtle graphics.

PROGRAMMING CONCEPTS: *Class, object, object-oriented programming, constructor, clones*

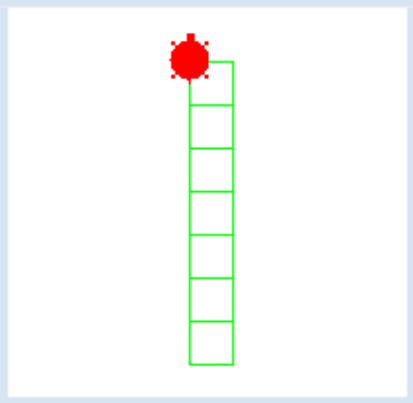
CREATING A TURTLE OBJECT

The turtle was previously used as an anonymous object which we did not use a name for. If you want to use multiple turtles at the same time, you must give every turtle an identity by naming it. You can then use the name as a variable name.

With the statement **maya = Turtle()** you create a turtle named *maya*.
With the statement *pepe = Turtle()* you create a turtle named *pepe*.

You can control the named turtles with the commands you already know, but you always have to say which turtle you mean. Put the turtle name first, followed by a point, and finally the command, for example **maya.forward(100)**.

In the first example *maya* draws a ladder. You do not need the line *makeTurtle()* anymore since you are creating the turtle yourself.



```
from gturtle import *

maya = Turtle()
maya.setColor("red")
maya.setPenColor("green")
maya.setPos(0, -200)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
    maya.forward(50)
```

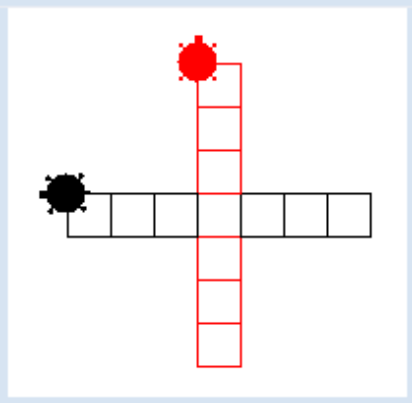
MEMO

Similar objects are grouped into classes. An object of a class is made (we also say "instantiated") by using the class name with a set of parentheses. We call this the **constructor** of the class. In the future we will call functions that belong to a particular class **methods**.

CREATING MORE TURTLE OBJECTS

Now you know all too well that you can use multiple turtles in the same program in the way previously described. If you want to create *maya* and *pepe*, then write
maya = Turtle() und **pepe = Turtle()**

These two turtles each end up in their own turtle window. You can put them into the same turtle enclosure by generating the enclosure as an object of the class *TurtleFrame*:
tf = TurtleFrame()
This object variable should be passed to the Turtle constructor while creating the turtles. At the same time that *maya* builds the same ladder as before, *pepe* builds a horizontal black staircase.



```
from gturtle import *

tf = TurtleFrame()

maya = Turtle(tf)
maya.setColor("red")
maya.setPenColor("red")
maya.setPos(0, -200)

pepe = Turtle(tf)
pepe.setColor("black")
pepe.setPenColor("black")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
        pepe.forward(50)
        pepe.left(90)
    maya.forward(50)
    pepe.forward(50)
```

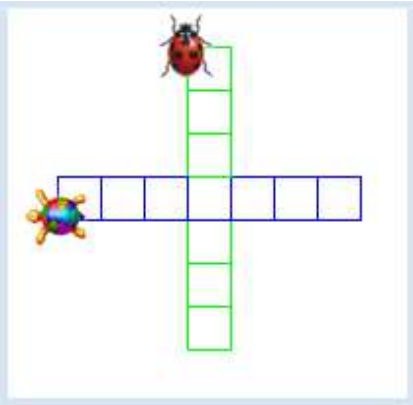
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

If you want to put multiple turtles into the same window you need to create a *TurtleFrame* and specify it as a constructor parameter for the turtle. The turtles do not run into each other but rather move (so to speak) over one another, whereby the turtle moving last always ends up on top of all the others.

TURTLE PARAMETERS

Turtle objects can also be used as function parameters. Because the same code is used to draw a single ladder for both turtles, it is easiest to define a function **step()**. As a (formal) parameter you can use any name you would like, for example, just *t*. You then call the function twice, once passing it **maya**, and the other time passing it **pepe**.



```
from gturtle import *

def step(t):
    repeat 4:
        t.forward(50)
        t.right(90)
    t.forward(50)

tf = TurtleFrame()

maya = Turtle(tf, "sprites/beetle.gif")
maya.setPenColor("green")
maya.setPos(0, -150)
pepe = Turtle(tf, "sprites/cuteturtle.gif")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    step(maya)
    step(pepe)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

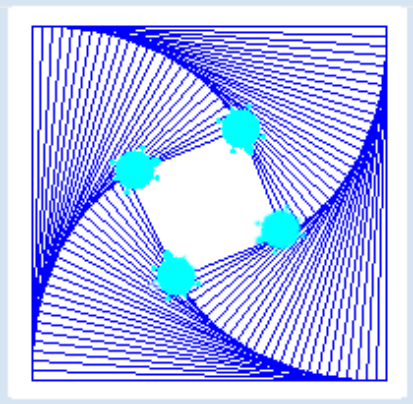
You can use your **own image** for each turtle if you specify the path to the image file while creating them. In the previous example, you used two image files *beetle.gif* and *cuteturtle.gif* which are located in the distribution of *TigerJython*.

MICE PROBLEM WITH A CLONED TURTLE

During the famous mice (or beetle) problem [**more...**] *n* beetles start at the corners of a regular *n*-gon and chase each other at a constant speed. The position of the beetles is fixed at equal steps of time and each one is rotated in the direction of the beetle in the next polygon corner. Afterwards, all of the beetles move forward at a steady increment.

You can nicely solve this problem by first drawing the polygon with the nameless (global) turtle and then by putting a cloned turtle at each corner. Here you choose a square and create the turtle clones t1, t2, t3, and t4 with **clone()**. A clone is a new turtle object with identical properties.

After that, you adjust their viewing direction in an endless loop with **setHeading()** and move them forward by 5. The drawing becomes especially nice if you draw out the connecting lines between each chasing turtle.



The easiest way to do this is to define the function **drawLine(a, b)**, with which the turtle *a* will draw a trail to turtle *b* and then go back again by using **moveTo()**.

```
from gturtle import *

s = 360

makeTurtle()
setPos(-s/2, -s/2)

def drawLine(a, b):
    ax = a.getX()
    ay = a.getY()
    ah = a.heading()
    a.moveTo(b.getX(), b.getY())
    a.setPos(ax, ay)
    a.heading(ah)

# generate Turtle clone
t1 = clone()
t1.speed(-1)
forward(s)
right(90)
t2 = clone()
t2.speed(-1)
forward(s)
right(90)
t3 = clone()
t3.speed(-1)
forward(s)
right(90)
t4 = clone()
t4.speed(-1)
forward(s)
right(90)
hideTurtle()

repeat:
    t1.setHeading(t1.towards(t2))
    t2.setHeading(t2.towards(t3))
    t3.setHeading(t3.towards(t4))
    t4.setHeading(t4.towards(t1))

    drawLine(t1, t2)
    drawLine(t2, t3)
    drawLine(t3, t4)
    drawLine(t4, t1)

    t1.forward(5)
    t2.forward(5)
    t3.forward(5)
    t4.forward(5)
```

MEMO

By using **clone()** you are creating a new turtle from the global turtle, so it will have the same position, the same viewing direction, and the same color (if you are using a custom turtle icon, it will have the same image) [**more...**] .

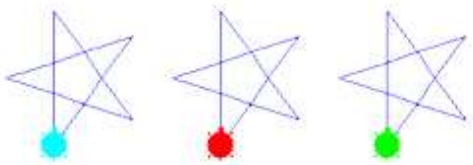
The function *drawLine()* can be simplified if you save the position and orientation of the turtle with *pushState().* The state can then be retrieved again with *popState()*:

```
def drawLine(a, b):
    a.pushState()
    a.moveTo(b.getX(), b.getY())
    a.popState()
```

The emerging chasing curves can be calculated mathematically (**see here**).

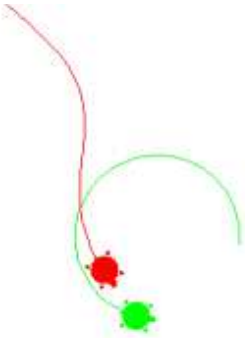
EXERCISES

1. Three turtles should alternately, point by point, draw a five-pointed star. The turtles should be colored cyan (the standard color), red, and green. The turtle's color can be specified as an additional parameter of the constructor.



2. A green mother turtle constantly draws a circle with a green pen color. At first, the red child turtle is far away from the mother turtle and then moves with the red pen towards the mother.

   (The child turtle named *child* can determine the direction of the mother using *direction = child.towards(mother)*

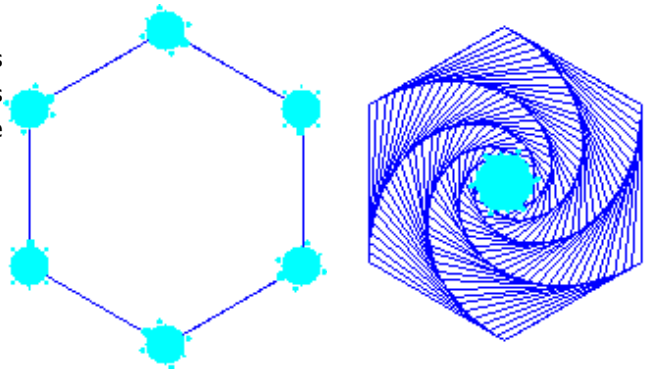


3. 

   Laura draws (not filled) squares. After each square is drawn, a second turtle jumps in and colors it green.

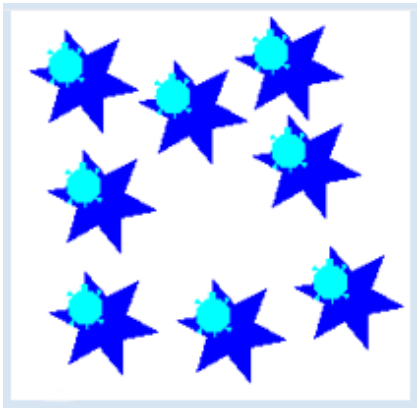
   Use different turtle images for the two turtles. The images available in *tigerjython2.jar* are *beetle.gif*, *beetle1.gif*, *beetle2.gif,* and *spider.png*. You can also use your own images. If you do, you have to save it in the subdirectory *sprites*, which is located in the same directory as your program.

4. Create chasing graphics for 6 turtles that start their chase at the corners of a regular 6-gon, similar to the example "beetle problems".



EXTRA MATERIAL

CREATING TURTLES WITH A MOUSE CLICK



With every mouse click your program will create a new turtle that draws a star, at the location of the mouse cursor, independently of other existing turtles. In this example you can experience the full scope and the beauty of object-oriented programming as well as event control.

To process the mouse click, you define the function **drawStar()**. In order for the function to be called by your system when you press the left mouse button, you must use the parameter **mouseHit** in the constructor of the turtle frame and give it the name of this function.

```
from gturtle import *

def drawStar(x, y):
    t = Turtle(tf)
    t.setPos(x, y)
    t.fillToPoint(x, y)
    for i in range(6):
        t.forward(40)
        t.right(140)
        t.forward(40)
        t.left(80)

tf = TurtleFrame(mouseHit = drawStar)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Objects that have the same capabilities and similar properties are defined as classes in OOP. You can create individual objects with the constructor (**instances**).

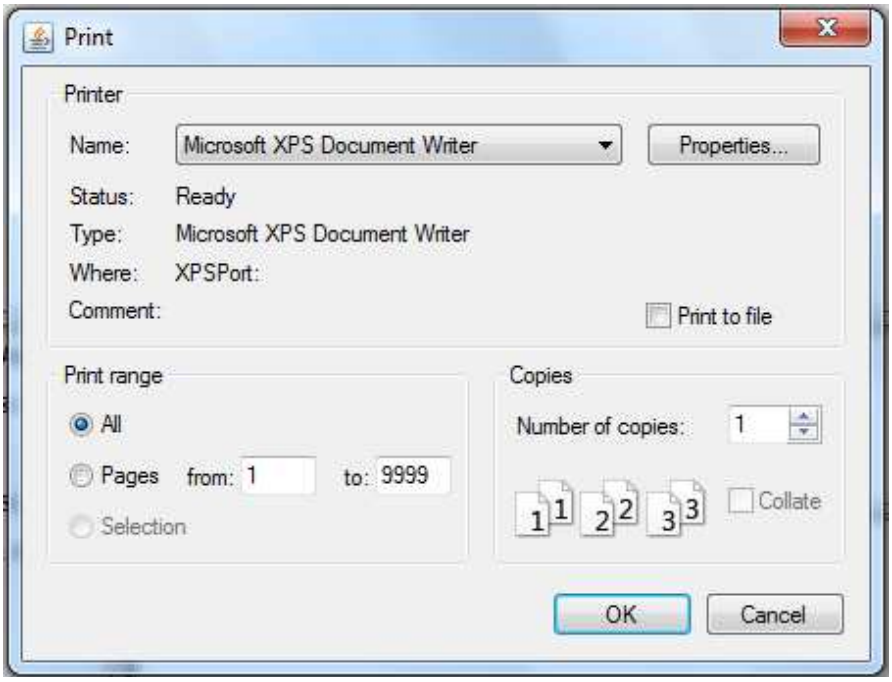
In order to process a mouse click, you have to write a function with a name of your choice (including two parameters, *x* and *y*) and then pass this name to the parameter *mouseHit* in the constructor of the *TurtleFrame*. *x* and *y* provide the coordinates of the mouse click.

2.12 PRINTING

INTRODUCTION

To achieve higher precision you can draw graphics on a printer since printers usually have a much higher resolution, for example 1200x1200 dpi, where a typical screen resolution is around 100 dpi. The printing of GPanel graphics works in a way where graphic operations are printed onto paper instead of being displayed on the screen. For this, you can define a function with any name that contains all of the instructions for creating the image. When it is called directly, the image appears on the screen. To print, call *printerPlot()* with the function name.

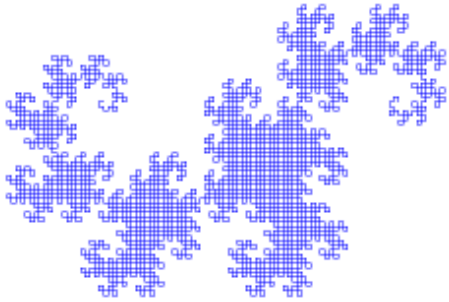
A printer dialog opens where you can select the printer and set its properties. You can then also print on virtual printers to create a graphic file in a high resolution format (for example TIFF or EPS).



PROGRAMMING CONCEPTS: *High resolution graphics*

A NON-FIRE-BREATHING DRAGON

To show a fun example of high-resolution printing, let the turtle draw a complicated picture called the dragon curve. Although you could also make this curve by folding a strip of paper, it is much easier with computer graphics. However, the implementation of the folding instructions in an algorithm is not entirely trivial. Since we are dealing with printing here, we simply use a pre-defined function to draw the curve, *figure(s, n, flag)*.



At least you can see that the curve is defined recursively and calls itself twice, although with a lower order *n-1*. Additionally, the function uses a parameter flag that can be 1 or -1 and that determines the drawing direction.

To print the image, draw it in the function *doIt()*. This function cannot have parameters. When you call *doIt()* the drawing appears on the screen, but when you pass the name *doIt* to the *printerPlot()* command the drawing is printed (without showing the turtle).

```
from gturtle import *
import math

nbGenerations = 12

def doIt():
    rt(90)
    figure(300, nbGenerations, 1)

def figure(s, n, flag):
    if n == 0:
        fd(s);
    else:
        alpha = 45
        if flag == 1:
            alpha = -alpha
            flag = -flag
        lt(alpha)
        figure(s / math.sqrt(2), n - 1, -flag)
        rt(2 * alpha)
        figure(s / math.sqrt(2), n - 1, flag)
        lt(alpha)

makeTurtle()
ht()
setPos(-100, 100) # screen
doIt()
setPos(100, 0)      # printer
printerPlot(doIt)
```

MEMO

You have to position the drawing on the sheet properly by using *setPos()*. Depending on the size of the turtle window and the printer being used, this position can change. When calling *printerPlot()* you can also specify a scaling factor *k* in *printerPlot(doIt, k)*. The image enlarges when k > 1 and shrinks when k < 1.

EXERCISES

1. Joshua Goldstein suggests using pairs of move/turn commands to create nice pictures. A step consists of the commands forward(s) and right(a)

   Draw and print the following Goldstein figures:
   a. 31 steps where s = 300, a = 151°
   b. 142 steps where s = 400, a = 159.72°
   You must provide the positioning yourself!

2. A step can also consist of two move/turn pairs.
   Draw and print the following Goldstein figure:
   37 steps where s = 77, a = 140.86° and s = 310, a = 112°

3. Draw and print the Goldstein figure with three move/turn pairs:
   47 steps where s = 15.4, a = 140.86 ° and s = 62, a = 112° and s= 57.2, a = 130°

Documentation Turtle Graphics

Module import: from gturtle import *

Function	Action
makeTurtle()	creates a (global) turtle in a new window and defines all global commands
makeTurtle(color)	same, but creates a turtle with a specified color
makeTurtle("sprites/turtle.gif")	same, but creates a turtle with given sprite image
t = Turtle()	creates a turtle object t
tf = TurtleFrame()	creates a graphics windows where several turtles may live together
tf = TurtleFrame(title)	dasselbe, aber mit gegebenen Titel
t = Turtle(tf)	creates a turtle in the given turtle frame
clone()	creates a turtle clone (same color, position, viewing direction)
isDisposed()	returns True, if the window is closed
putSleep()	pauses program execution until wakeUp() is called
wakeUp()	resumes paused program execution
enableRepaint(False)	disables automatic screen rendering
repaint()	renders the screen manually (after disabling automatic rendering)
savePlayground(fileName, format)	copies the playground into a image file (format: "png" or "gif"). Returns True, if successful; otherwise False

Movements

back(distance), bk(distance)	moves the turtle backwards for given distance (in turtle coordinates)
forward(distance), fd(distance)	moves the turtle forwards for given distance (in turtle coordinates)
hideTurtle(), ht()	hides the turtle (speeds-up the drawing)
home()	puts the turtle into the middle of the window with upward direction
left(angle), lt(angle)	turns the turtle to the left (in degrees)
penDown(), pd()	activates the pen (trace becomes visible)
penErase(), pe()	sets the pen color to the background color
leftArc(radius, angle)	moves the turtle on a left oriented arc with with given radius and sector angle
leftCircle(radius)	moves the turtle on a left oriented circle with given radius (in turtle coordinates)
penUp(), pu()	deactivates the pen (trace becomes invisible)
penWidth(width)	selects the pen width (in pixels)
right(angle), rt(angle)	turnes the turtle to the right (in degrees)
rightArc(radius, angle)	moves the turtle on a right oriented arc with with given radius and sector angle
rightCircle(radius)	moves the turtle on a right oriented circle with given radius (in turtle coordinates)
setCustomCursor(cursorImage)	selects image file used as mouse cursor
setCustomCursor(cursorImage, Point(x, y))	selects image file use as mouse cursor and defines the hotspot relative to the picture
setLineWidth(width)	sets the pen width (in pixels)
showTurtle(), st()	shows the turtle
speed(speed)	sets the speed of the turtle movement
delay(time)	stops the program for the given amount of time (in milliseconds)
wrap()	turtle positions outside the window are mapped inside the window (torus symmetry)
clip()	turtles outside the window are invisible
getPlaygroundWidth()	returns the width m of the turtle Playground (turtle coordinates -m/2...m/2)
getPlaygroundHeight()	returns the height n of the turtle Playground (turtle coordinates -n/2...n/2)

Positioning

direction(x, y)	returns the angle to turn to the position (x, y) zurück
direction(coords)	same, but coordinates given as list, tuple or complex
direction(turtle)	returns the angle to turn to the position of another turtle
distance(x, y)	returns the distance of the turtle to point(x, y)
distance(coords)	same, but coordinates given as list, tuple or complex
distance(turtle)	returns the distance to another turtle
getPos()	returns the current position (list)
getX()	returns the current x-coordinate
getY()	returns the current y-coordinate
heading()	returns the current viewing direction (in degrees, clockwise to the north)
heading(degrees)	sets the viewing direction (in degrees, zero to the north, positive clockwise)
moveTo(x, y)	moves the turtle to the given coordinates by drawing the trace
moveTo(coords)	same, but coordinates given as list, tuple or complex
setHeading(degrees), setH(degrees)	sets the viewing direction (in degrees, zero to the north, positive clockwise)
setRandomHeading()	sets the viewing direction to a random value 0 ... 360°
setPos(x, y)	moves the turtle to the given coordinates without drawing the trace
setPos(coords)	same, but coordinates given as list, tuple or complex
setX(x)	sets the turtle to given x-coordinate
setY(y)	sets the turtle to given y-coordinate
setRandomPos(width, height)	sets the turtle position to a random value in the given rectangle
setScreenPos(x, y)	sets the turtle position to the given screen coordinates
setScreenPos(Point(x, y))	sets the turtel position to the given point
towards(x, y)	returns the direction (in degrees) to the given coordinates
towards(coords)	same, but coordinates given as list, tuple or complex
towards(turtle)	returns the direction to another turtle
toTurtlePos(x, y)	returns a list of the turtle coordinates of the given pixel coordinates
toTurtlePos(Point(x, y))	returns a list of the turtle coordinates of the given point
pushState()	saves the current turtle state on a stack (first-in-last-out)
popState()	sets the turtle state to the last element of the stack and removes the state from the stack
clearStates()	removes all elements from the state stack

Colors

askColor(title, defaultColor)	opens a color selection dialog and returns the selected color
clear()	erases the traces and hides all turtles, but let them where they are
clear(color)	erases the traces, hides all turtles and paint the background with the given color
clean()	erases everything and puts all turtles to the home position
clean(color)	erases everything, puts all turtles to the home position and paint the background with the color
dot(diameter)	paints a filled circle with given radius using the current pen color
openDot(diameter)	paints a non-filled circle with given radius using the current pen color
fill()	fills a closed area around the current turtlle position with the current fill color
fill(x , y)	fills a closed area around the given position with the current fill color
fill(coords)	same, but coordinates given as list, tuple or complex
fillToPoint()	fills continuously from the current turtle position
fillToPoint(x , y)	fills continuoiusly from the given point(also list, tuple, complex)
fillToHorizontal(y)	fills continuously die area between the horizontal line and the turtle position

fillToVertical(x)	fills continuously die area between the veritcall line and the turtle position
fillOff()	terminates the fill mode
getColor()	returns the turtle color
getColorStr()	returns the turtle color as X11 color string
getFillColor()	returns the fill color
getFillColorStr()	returns the fill color as X11 color string
getPixelColor()	returns the color of the pixel (background or trace) at the current turtle position
getPixelColorStr()	returns the color as X11 color string of the pixel at the current turtle position
getRandomX11Color()	returns a random X11 color string
makeColor()	returns a color reference of given value. Value examples: ("7FFED4"), ("Aqua-Marine"), (0x7FFED4), (8388564), (0.5, 1.0, 0.83), (128, 255, 212), ("rainbow", n) with n = 0..1, light spectrum
setColor(color)	sets the turtle color
setPenColor(color)	sets the turtle's pen color
setPenWidth(width)	sets the turtle's pen width
setFillColor(color)	sets the turtle's fill color
startPath()	starts to register the turtle movement for a following fll operation
fillPath()	closes the fill operation at current turtle position and fills the path with the current flll color
stampTurtle()	creates an image of the turtle at the current position
stampTurtle(color)	creates an image of the turtle with given color at current position

Callbacks

makeTurtle(mouseNNN = onMouseNNN) use a comma separator to register more than one	registers the callback function onMouseNNN(x, y) that is called when a mouse event happens. Values for NNN: Pressed, Released, Clicked, Dragged, Moved, Entered, Exited, SingleClicked, DoubleClicked, Hit: Invocation in separate thread, HitX: same, but events are ignored until the previous callback returns
isLeftMouseButton(), isRightMouseButton()	returns True, if the event is caused by the left/right mouse button
makeTurtle(keyNNN = onKeyNNN)	registers the callback onKeyNNN(keyCode) that is called when a keyboard key is hit. Values for NNN: Pressed, Hit: Invocation in separate thread, HitX: same, but events are ignored until the previous callback returns. keyCode is a unique integer value that identifies the key
getKeyModifiers()	returns an integer code for special keyboard keys (shift, ctrl, etc., also combined)
makeTurtle(closeClicked = onCloseClicked)	registers the callback onCloseClicked() that is called when the title bar close button is hit. The window must be closed manually by calling dispose()
makeTurtle(turtleHit = onTurtleHit)	registers the callback function onTurtleHit(x, y) that is called when the turtle image is clicked
t = Turtle(turtleHit = onTurtleHit)	registers the callback function onTurtleHit(t, x, y) that is called when the image of turtle t is clicked

Text, Images and Sound

addStatusBar(20)	adds a status bar 20 pixels height
beep()	emits a short tone
playTone(freq)	plays tone mit given frequency (in Hz) and duration 1000 ms (blocking function)
playTone(freq, blocking=False)	same, but not-blocking function, used to play several tones at the same time
playTone(freq, duration)	plays tone with given frequency and given duration (in ms)
playTone([f1, f2, ...])	plays several tones in a sequence with given frequency and duration 1000 ms
playTone([(f1, d1), (f2, d2), ...])	plays serveral tones in a sequence with given frequency and given duration
playTone([("c",700), ("e",1500), ...])	plays serveral tones in a sequence with given (Helmholtz) pitch naming and duration. Supported are: great octave, one-line to three-line octave (range C, C# up to h'"

playTone([("c",700), ("e",1500), ...], instrument = "piano")	same, but selects instrument type. Supported are: piano, guitar, harp, trumpet, xylophone, organ, violin, panflute, bird, seashore, ... (see MIDI specifications)
playTone([("c",700), ("e",1500), ...], instrument = "piano", volume=10)	same, but selects sound volume (0..100)
label(text)	displays the given text starting at the current turtle position
printerPlot(draw)	prints the traces that are drawn in function draw
setFont(Font font)	defines the font used by label()
setFontSize(size)	defines the font size used by label()
getTextHeight()	returns the height of texts with current font (in pixels)
getTextAscent()	returns the ascender height of texts with current font (in pixels)
getTextDescent()	returns the descender height of texts with current font (in pixels)
getTextWidth(text)	returns the width of given text with current font (in pixels)
setStatusText(text)	shows the given text in the status bar. Any exiting text is erased.
setTitel(title)	shows the given title in the window title bar
img = getImage(path)	retrieve the image from the jar resource, from local drive or from a webserver
drawImage(img)	draws the given image into background with the image center at the current turtle position and rotated to the current turtle viewing direction.
drawImage(path)	loads an image (in png, gif or jpg format) from the local file system or from an URL and draws it at the current turtle position with the current viewing direction. For path = sprites/nnn an image from the TigerJython distribution is loaded

Dialogs

msgDlg(message)	opens a modal dialolg with an OK button and given message
msgDlg(message, title = text)	same with title text
inputInt(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns integer (the dialog is shown again, if no integer is entered). Cancel or Close terminate the program
inputInt(prompt, False)	same, but Cancel/Close do not terminate, but returns None
inputFloat(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns float (the dialog is shown again, if no float is entered). Cancel or Close terminate the program
inputFloat(prompt, False)	same, but Cancel/Close do not terminate, but returns None
inputString(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns string. Cancel or Close terminate the program
inputString(prompt, False)	same, but Cancel/Close do not terminate, but returns None
input(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns integer, float or string. Cancel or Close terminate the program
input(prompt, False)	same, but Cancel/Close do not terminate, but returns None
askYesNo(prompt)	opens a modal dialog with Yes/No buttons. Yes returns True, No returns False. Cancel or Close terminate the program
askYesNo(prompt, False)	same, but Close does not terminate, but returns None

chapter three

2D GRAPHICS & PICTURES

Learning Objectives

- You can create simple 2D graphics with geometric shapes.
- You know how to use keyboard and mouse inputs in the graphics window.
- You can display simple function graphs y = f(x) in the graphics window.
- You know that a digital image consists of colored pixels that are stored as numbers.
- You can load a digital image, alter it in specific ways, represent it on the screen, and save it.
- You can write keyboard- and mouse-controlled programs.
- You know how to generate random numbers and can apply this to random experiments.
- You can use input fields, buttons, and menus in your programs.

"A picture is worth a thousand words."

Old Proverb

3.1 COORDINATES

INTRODUCTION

You have already made your first experience drawing with the turtle on the computer. However, the turtle has its limits and so now you will get to know more flexible options of creating graphics output.

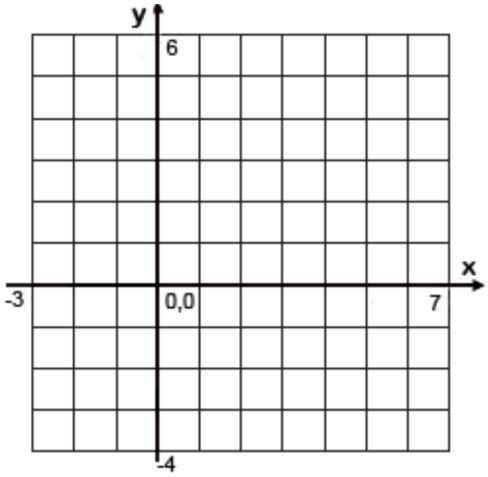
PROGRAMMING CONCEPTS: *Coordinate graphics, Cartesian coordinate system*

OPENING THE GRAPHIC WINDOW

The library (respectively the window for the graphics output) is called GPanel. This library is already installed in *TigerJython* but you must still specify that you want to use the GPanel and therefore start your program with an import. Then you can use **makeGPanel()** to create a new graphics window:

```
from gpanel import *
makeGPanel(-3, 7, -4, 6)
```

So far, the program does not do anything very exciting as it only shows a blank window that you can then close again. Your GPanel window is always square and uses an x-y-coordinate system, as you should know from mathematics:



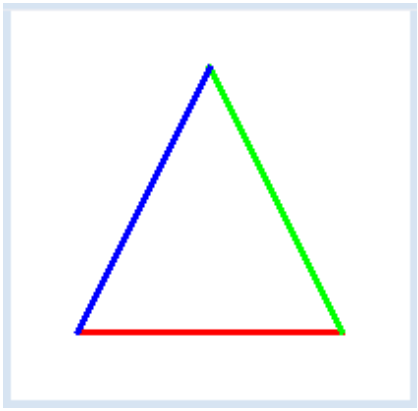
In this example you specify the x- and y-coordinate range with the four numbers -3, 7, -4, and 6. -3 is the x-coordinate on the left edge, 7 is the x-coordinate on the right edge, -4 is the y-coordinate on the bottom edge and 6 is the y-coordinate on the top edge.

MEMO

You can create a window with **makeGPanel().** You can define the desired area of the coordinate system with four numbers: *makeGPanel(xmin, xmax, ymin, ymax)*

You can also specify a window title as the first parameter:
makeGPanel(title, xmin, xmax, ymin, ymax)

DRAWING LINES



Once the window is open, you can begin drawing in it just as you like. There are a number of useful functions that can help. For example, with **line()** you can draw a line, and with **setColor()** you can alter the color. You can then, for example, draw a colorful triangle.

For each line, you first specify the x- and y-coordinates of the start point, then the x- and y-coordinates of the end point. The vertices should have the following coordinates: (1, -1) (5, -1) (3, 3). Of course you can only see the triangle if you choose the coordinate system appropriately. Undistorted drawings will only appear if the coordinate system is the same length in both the x- and y-direction.

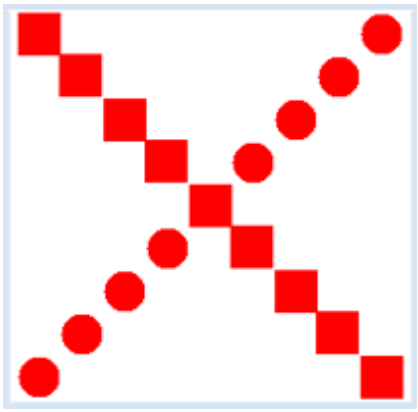
```
from gpanel import *

makeGPanel("My window", 0, 6, -2, 4)

lineWidth(3)
setColor("red")
line(1, -1, 5, -1)
setColor("green")
line(5, -1, 3, 3)
setColor("blue")
line(3, 3, 1, -1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can specify the width of the line in pixels using the function **lineWidth()**.

CIRCLES AND RECTANGLES



GPanel can draw not only lines, but also circles, ellipses, rectangles, triangles, and arcs. It can even write out texts. You can draw a filled circle with the command *fillCircle(radius)*. But before you draw a circle, you need to position the graphics cursor using *move(x, y)* in order to define the center point.

fillRectangle(length, width) draws a rectangle with its center at the position of the graphics cursor. In our example, we draw several squares and circles using a *while* loop.

```
from gpanel import *

makeGPanel(0, 20, 0, 20)

setColor("red")
x = 2
y = 2
while y < 20:
    move(x, y)
    fillCircle(1)
    move(x, 20 - y)
    fillRectangle(2, 2)
    x = x + 2
    y = y + 2
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can draw many different figures with GPanel. Here are the most important commands:

point(x, y)	A point
line(x1, y1, x2, y2)	A line
rectangle(width, height)	A rectangle (width, height)
fillRectangle(width, height)	A filled rectangle
rectangle(x1, y1, x2, y2)	A rectangle (vertices)
fillRectangle(x1, y1, x2, y2)	A filled rectangle
fillTriangle(x1, y1, ..., y3)	A triangle (vertices)
circle(r)	A circle with radius *r*
fillCircle(r)	A filled circle
ellipse(a, b)	An ellipse with axes a, b
fillEllipse(a, b)	A filled ellipse
arc(r, a, b)	An arc
text("t")	Writes the text *t*
move(x, y)	Determines position

For circles, arcs, ellipses, text, and rectangles that are simply defined by length and width, you must first determine their position by setting the graphics cursor using *move()*.

GPanel knows the so-called ***X11 colors***. There are a few dozen color names that you can find on the Internet here: **http://cng.seas.rochester.edu/CNG/docs/x11color.html**. You can choose all of these colors using ***setColor(color)***.

EXERCISES

1. Draw a similar figure:

2. What does a rainbow actually look like? Let the computer draw you a rainbow. Use the function *circle(r)* so that only the upper part of the circle is visible.

3.2 FOR LOOPS

INTRODUCTION

You often have to count during repetition. For that, you need a variable in the repetition block that changes by a certain value in every iteration of the loop. It is easier to do this using a *for structure* than it is with a *while structure*. You must first understand the *range() function*. In the simplest case, *range()* has a single parameter (also called *stop value*) and provides a sequence of natural numbers that starts with 0 and ends with the last number before the stop value.

You can try this out with a few examples. If, for example, you execute a program with the single statement *print range(10),* the numbers [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] are written in the output window. Try it out with a few different parameters. As you can see, in our example, the stop value 10 is not included in the list; it rather indicates how many list elements there are.

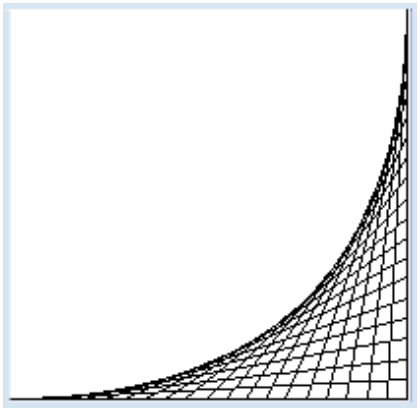
PROGRAMMING CONCEPTS: *Iteration, for structure, nesting of for loops*

FAMILY OF LINES

You can draw a cool curve with 20 lines using this **for structure**.

```
from gpanel import *

makeGPanel(0, 20, 0, 20)

for i in range(21):
    line(i, 0, 20, i)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)



MEMO

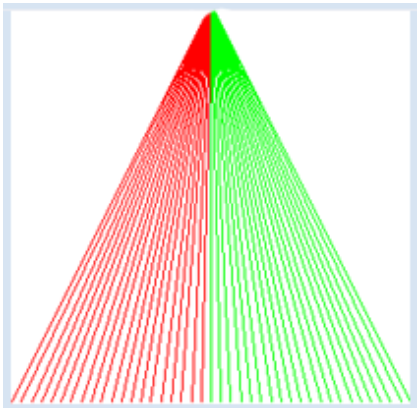
The statement **for i in range(n)** runs through the numbers from 0 to n-1, so in other words, a total of *n* numbers. The places of consolidation of the lines form a *quadratic Bézier curve*.

RANGE() WITH TWO PARAMETERS

The range function can also have **two parameters**. In this case, the first parameter is the *start value* of the list and the second is the *stop value,* which is, however, not included in the list.

If, for example, you write print *range(2, 9)*, the numbers [2, 3, 4, 5, 6, 7, 8] are written in the output window. Try it out using a few other parameters.

Using the following program, you draw lines in two colors with the start points on the x-axis from the coordinates -20 to 20. The endpoint of all lines is the point (0, 40).



```
from gpanel import *

makeGPanel(-20, 20, 0, 40)

for i in range(-20, 21):
    if i < 0:
        setColor("red")
    else:
        setColor("green")
    line(i, 0, 0, 40)
```

MEMO

The loop **for i in range(start, stop)** with integer *start* and *stop values* begins at i = start and ends at i = stop - 1, where the loop counter *i* is increased by 1 each time it runs through the loop. Thereby you need to make *start* smaller than *stop*, otherwise the loop will never run.

RANGE() WITH THREE PARAMETERS

You can even call the range function with **three parameters.** In this case, the first parameter is the *start value* of the list, the second is the *stop value*, and the third is the change in value from one element to the next. This will help you to adjust the step size, which was previously always 1, to any situation.

If, for example, you write print *range(2, 15, 3)*, the numbers [2, 5, 8, 11, 14] are written in the output window.

In the adjacent graphic you draw a pyramid standing on its peak with filled rectangles. The smallest rectangle has a width of 2, and the biggest has a width of 40.

```
from gpanel import *

makeGPanel(0, 40, 0, 40)

setColor("red")
y = 1
for i in range(2, 41, 2):
    move(20, y)
```

```
    fillRectangle(i, 1.5)
    y = y + 2
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

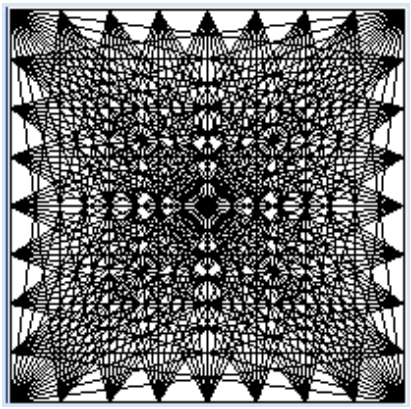
MEMO

The loop **for i in range(start, stop, step)** begins with *i* = *start* and ends at a value that is less than *stop*. *i* is increased by *step* each time the program runs through the loop. You can also choose negative numbers for the values *start, stop* and *step*.

I *step* is negative, *i* is reduced by *step* at every iteration; the last value is greater than *stop*.

NESTED *FOR* LOOPS (Moiré)

Closely drawn lines positioned above one another can produce an optical effect called the Moiré pattern. In a square, you draw lines from 10 points on the bottom edge to each of 10 points on the upper edge. Then you do the same from the left to the right edge.



```
from gpanel import *

makeGPanel(0, 10, 0, 10)

for i in range(0, 11):
    for k in range (0, 11):
        line(i, 0, k, 10)
        delay(100)
for i in range(0, 11):
    for k in range (0, 11):
        line(0, i, 10, k)
        delay(100)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

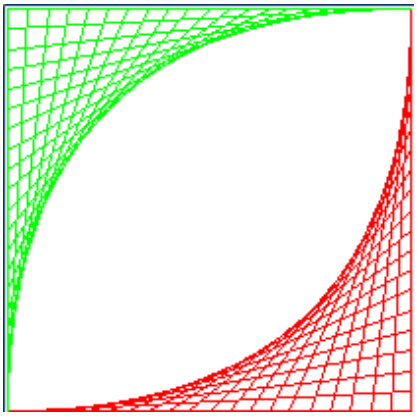
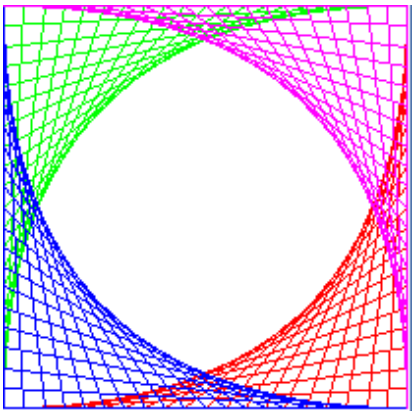
The program might not be very easy to understand, but it is important. At best, you can assume that the loop variable *i* of the outer loop has a constant value (initially 0) . The inner loop runs with this value for the values k = 0 up to and including 10. Then *i* is set to 1 and the inner loops run again with this value *i*, etc.

When using the command **delay**(millisec), the program waits for the given number of milliseconds so that you can observe how the pattern emerges gradually.

EXERCISES

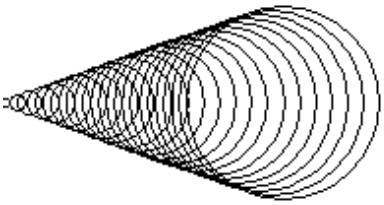
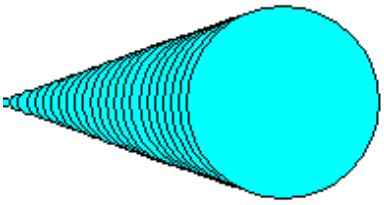
1. You will get an even nicer graphic than the one seen in Example 1 if you use colors. Draw a

second family of lines with a different color (Figure a).

(a)  (b) 

The blue family of lines (Figure b) is drawn with *line(i, 0, 0, 20 - i)*. Can you also program the purple one?

2. Draw a family of circles.

You can draw the colored family of circles with the following procedure: First draw a filled circle with the radius *y*, then choose the color black and draw another circle with the same radius:

```
setColor("cyan")
fillCircle(y)
setColor("black")
circle(y)
```

3. In Example 3 we drew a pyramid standing upside down. Draw a "real" pyramid with three colors. In order to do so, you can use a *for* loop that counts down.

.

3.3 STRUCTURED PROGRAMMING

INTRODUCTION

The concept of variables is very important to programming. Therefore, you need to give a special effort in order to understand it as thoroughly as possible. You already know that a variable is a memory slot that is addressed with a name and that holds a value. You also know that parameters can be understood as "volatile" memory slots which, when their function is called, receive a value that the function can then access during its execution.

PROGRAMMING CONCEPTS: *Constants, procedural programming, reusability*

A MOSAIC OF 10X10 STONES

You have the task of creating a beautiful colored mosaic out of square stones. You get different colored mosaic stones with the side length 10, and you should put them together exactly side by side on a canvas with the size 400x400. You feel a bit lazy, so you are going to leave the task up to the computer. You tell it to place the stones with **random colors** line by line. Use **delay(1)** to create a short pause after each stone is laid so that you can watch the computer making the mosaic.

```
from gpanel import *
makeGPanel(0, 400, 0, 400)

for y in range(0, 400, 10):
    for x in range(0, 400, 10):
        setColor(getRandomX11Color())
        move(x + 5, y + 5)
        fillRectangle(10, 10)
        delay(1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

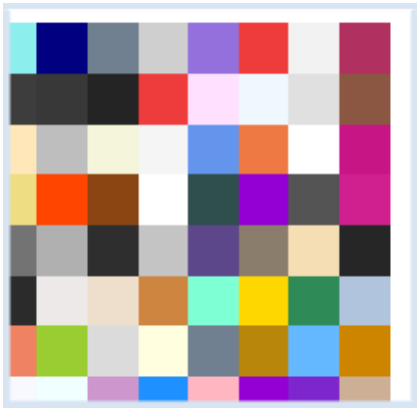
Whenever a grid needs to be run through, two nested *for* loops are best suited. Think about why precisely the stones are placed line by line from bottom to top. You need a shift at *move()* because the stones are anchored/established in the middle.

The method **getRandomX11Color()** gives back (as a word) one of the colors from the X11 color palette, which you can then pass on to *setColor()*. You can first call the function *getRandomX11Color()*, and then the function *setColor()*.

MAGIC NUMBERS

Two weeks later, you receive a delivery of stones that are five times larger with a side length of 50. The computer should lay them on the same canvas again. What do you need to do to adjust the program? You check out the previous code, but of course you no longer understand what each particular line means.

You think: anywhere there is a 10, the number should be changed to 50. So you do this, but soon realize you were wrong. Unfortunately the mosaic no longer covers the entire canvas.



Now you have to go back through the code line by line in order to find the error. If this is necessary to adapt the program to a new situation (**reuse**), then your program was correct but just poorly written. You should get into the habit of having a **good programming style**, so that you can easily adjust programs to (fit) new situations.

How can you proceed? Instead of writing the stone size as a **fixed number** in the code, define a variable *size*, that can be used in place of the fixed number wherever the stone size plays a role. In order to **structure** the program even better, you should also write a separate function **drawStone()** for the placement of the stone.

```
from gpanel import *

def drawStone(x, y):
    setColor(getRandomX11Color())
    move(x + size/2, y + size/2)
    fillRectangle(size, size)

makeGPanel(0, 400, 0, 400)

size = 50

for x in range(0, 400, size):
    for y in range(0, 400, size):
        drawStone(x, y)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The use of fixed numerical values throughout the program results in poorly reusable programs. You should instead define variables and use them in place of numbers. To indicate that these should never change, sometimes you write these variables in **capital letters** and call them **constants**.

A variable that is defined in the main block can also be read in each function. Therefore, we also call it a **global variable**.

For longer self-contained actions, the code should be put into separate functions. This has several advantages: the first is that you can recognize the function name and what it should do, the second is that you can call them several times without having to rewrite the code, and the third is that the program becomes clear and comprehensible. This type of programming is called **structured programming** (or **procedural programming**) and it is an important trait of a good programming style.

EXERCISES

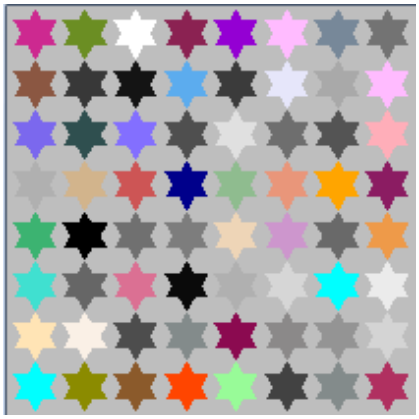
1. As you can find out using the Pythagorean theorem, the command:

   ```
   fillTriangle(x - math.sqrt(3)/2 * r, y - r/2,
                x + math.sqrt(3)/2 * r, y - r/2,
                x, y + r);
   ```

   draws an equilateral triangle with the center point at (x, y) and the radius *r*. Verify this in a GPanel with the coordinate system -1..1 for both axes, which draws a triangle with its center at the origin.

2. Use the code from exercise 1 and define a function *star(x, y, r)*, which draws a star using two equilateral triangles with the center point (x, y) and the size *r*. Try it out and draw a few stars.

3. Enhance the function *star()* with a parameter that sets the color of the star. Draw a star mosaic on a grey background with 50x50 stars. Keep the size of the stars consistent.
   Also, make sure that none of the stars are drawn with the background color.



3.4 FUNCTIONS WITH A RETURN VALUE

INTRODUCTION

You already know how to define a function with or without parameters and how to call it. From mathematics you probably know that there, functions are understood as something slightly different. In mathematics, a function y = f(x) has an independent variable *x*. For each value of *x*, the function returns a value of the dependent variable *y*. One example is the quadratic function: $y = x^2$. x = 0, 1, 2, 3 results in the square numbers 0, 1, 4, 9.

You can also define functions in *Python* that calculate a value and then "return" it as a variable.

PROGRAMMING CONCEPTS: *Return value of a function, discretization*

THE KEYWORD RETURN

You can define a function *squarenumber(x)* that calculates a square number *x* * *x* for a given parameter *x*, just as in mathematics. The return occurs by using the keyword *return*. You can then draw the graph of the function in a GPanel. For the graphics you best use *draw(x, y),* which draws a line segment from the last position of the graphics cursor to (x, y) and then places it there. After the GPanel window appears, the graphics cursor is at (0, 0). You must first set it to the starting point of the image using *move()*, otherwise you will get an incorrect starting line.

```
from gpanel import *
makeGPanel(-25, 25, -25, 25)

def squarenumber(x):
    y = x * x
    return y

for x in range(-5, 6):
    y = squarenumber(x)
    if x == -5:
        move(x, y)
    else:
        draw(x, y)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

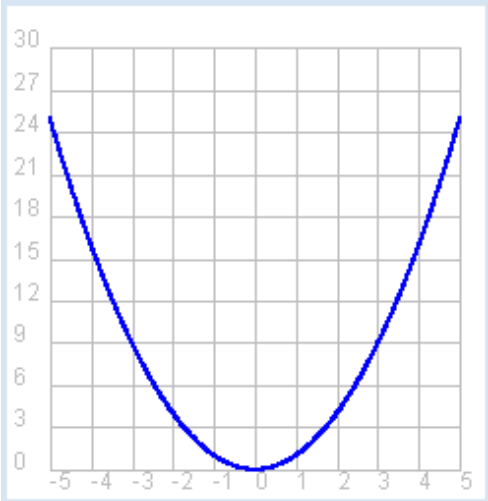
With **return**, a function can return a value to the caller and stop further processing. Just as in mathematics, a function cannot return multiple values [**more...**].

However, as you have seen, in contrast to mathematics, there are also functions in computer science that do not return a value but can still have an effect. Functions are even able to do both, cause something and return something [**more...**].

This graphical representation of the quadratic function is not very nice yet. In addition to the missing coordinate system, the graph's angular progression is also quite unpleasant. This is due to the fact that you only calculate the function at a few integer points that you connect with straight lines. This exposes an essential weakness of computer science compared to mathematics: Although the function delivers a *y* value for every value of the x-axis (for every real number), in computer science we can only calculate it at a finite number of points. We say that the continuous x-axis is **dissolved into discrete points**.

DECIMAL NUMBERS (FLOATS)

At least we can make the representation a bit nicer if we choose to calculate points that are close to each other on the x-axis. For example, you can run through the range of -5 to 5 in hundredth steps. To make it even better, you can draw a coordinate grid.



Unfortunately, you can only run *for* loops with integer values in *Python*. If you need a better resolution, you can use a *while* loop. This way, we can add 0.01 to the x coordinate at every step. Now *Python* does no longer consider x as an integer, but as a decimal number (float).

```
from gpanel import *
makeGPanel(-6, 6, -3, 33)
setColor("gray")
drawGrid(-5, 5, 0, 30)

def squarenumber(x):
    y = x * x
    return y

setColor("blue")
lineWidth(2)
x  = -5
while x < 5:
    y = squarenumber(x)
    if x == -5:
        move(x, y)
    else:
        draw(x, y)
    x = x + 0.01
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In *Python*, decimal numbers are called **float**. Unlike in mathematics, decimal numbers in computer science always have a certain (finite) number of digits. In *Python* there are about 14 digits (in other programming languages such numbers are called **double**). One example in computer programming is that you can never exactly specify the number π (which is of course an infinite decimal fraction), because you would only get a precision of around 14 digits with a float.

If you need a coordinate grid, you can do the following:

- Expand/Enlarge the coordinate range left and right, and up and down by 10% (instead of -5 to 5 use -6 to 6, instead of 0 to 30 use -3 to 33)
- Call *drawGrid()* with 4 parameter values, matching the coordinate range that you are actually using. This results in 10 coordinate fields.

EXERCISES

1. Define the function *mean(a, b)* that returns the arithmetic mean of the two parameters. Test it out with the console.

2. Examine the behavior of the function y = cos(x). How is it different from y = sin(x)?

3. Display the graph of the function y = sin(5x) in a GPanel, for a range between 0 and 2π with a resolution of 0.01 (you can get π using math.pi). Pick a value other than 5 within the sine function. What is the connection between this number and the graph?

4. Define the function f(x) = 1 / sin(x) and represent it in a GPanel for the range -5...5 (for both axes) with a solution of 0.001. Also draw the coordinate axes with a different color. Do you find anything interesting about this?

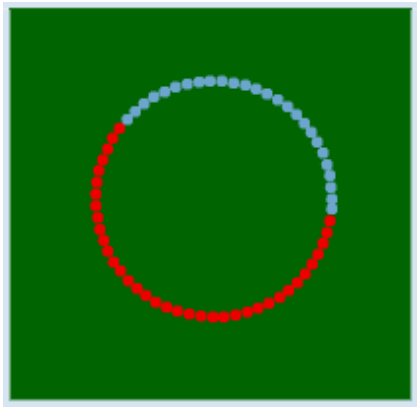
3.5 GLOBAL VARIABLES, ANIMATIONS

INTRODUCTION

Computer graphics are frequently used to represent time-varying content. For example, you can simulate a process from physics or biology, or create a computer game. We generally call such a program an **animation**. In order to show a temporal sequence, new images are drawn one after another, always after an equal step in time, called an **animation step.**

PROGRAMMING CONCEPTS: *Global variables, side effects, double buffering*

CHANGING GLOBAL VARIABLES



You want to illustrate a ball that moves on a circle. You will get a circular motion with a radius of 1 by calculating the x-coordinate using the cosine function with an increasing parameter *t* (corresponding to advancing time), and the y-coordinate with the sine function, thus x = cos(t) and y = sin(t). If you want a different radius, you have to multiply both values with the radius.

With the function **step()**, the situation for each animation step is drawn. Once the ball has made a full circle, the color should change.

It is common to introduce an endless **animation loop** in the main program that repeatedly calls *step()*. By incorporating a delay, you can change the speed of the animation. When using *step()*, the **global variable t** should increase at each step and reset to 0 once it has reached 2P, and the color should change.

```
import math
from gpanel import *

def step():
    global t
    x = r * math.cos(t)
    y = r * math.sin(t)
    t = t + 0.1
    if t > 6.28:
        t = 0
        setColor(getRandomX11Color())
    move(x, y)
    fillCircle(10)

makeGPanel(-500, 500, -500, 500)
bgColor("darkgreen")

t = 0
r = 200

while True:
    step()
    delay(10)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Python prohibits changing the value of global variables in functions. We can bypass this by providing the variable with the keyword **global** in the function.

The identifier *global* poses risks: any function can not only change a variable designated as global, but can also create it, as the following example shows:

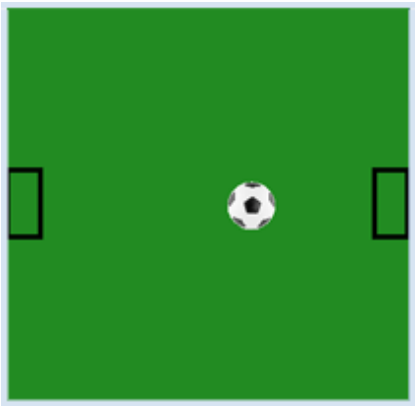
```
def set():
    global a
    a = 2

def get():
    print "a =", a
set()
get()
```

Since *set()* generates a variable *a* which is visible throughout the entire program, we also say that the function *set()* has **side effects**. Also note how nicely several things, one after another, can be written into the console using comma separation in a print statement. In the output the comma is replaced by a space.

THE TRICK WITH THE EVEN TICK

The animation loops should run in time ticks that are as even as possible, i.e. with the desired animation period, otherwise the movement will be jerky. With **step()** each new animation state is set up. Depending on the situation, this can take different amounts of time to complete, as the program may not always run the same parts of the code, and also because the computer may be busy with other tasks in the background, which may delay the execution of the *Python* code. To **compensate** for the *step()* of varying length, the following trick is used, which you could also have figured out yourself: before **calling step()**, you keep track of the current time using the variable *startTime*. After returning from *step()* you wait in a waiting loop until the **difference** between the new time and the start time reaches the animation period.

The program moves a soccer ball from goal to goal. For this, you use an image **football.gif** hat is located in the directory *sprites* of the *TigerJython* distribution. You can also take your own picture by copying the file into an appropriate directory on your computer and passing the file path as a parameter in *image()* (absolute or relative to the directory where your program is located).



```
from gpanel import *
import time

def step():
    global x
    global v
    clear()
    lineWidth(5)
    move(25, 300)
    rectangle(50, 100)
    move(575, 300)
    rectangle(50, 100)
```

```
    x = x + v
    image("_sprites/football.gif", x, 275)
    if x > 500 or x < 50:
        v = -v

makeGPanel(0, 600, 0, 600)
bgColor("forestgreen")
enableRepaint(False)

x = 300
v = 10

while True:
    startTime = time.clock()
    step()
    repaint()
    while (time.clock() - startTime)  < 0.020:
        delay(1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Using *time.clock()* you can get the current time as a decimal number. The given value is dependent on the computer (processor time, or the time since the first call of *clock()*). But since you only need the time difference, it does not matter. Save the time before calling *step()* and wait at the end of the animation loop with a *delay(1)*, until the difference between the current time and the start time reaches the animation period (in seconds). Remember this trick, for you will use it for many processes that ought to run as periodically as possible.

Every graphics command is immediately visible in the GPanel window. Deleting with **clear()**

while animating will briefly show an empty graphics window, which can lead to a flickering effect. To avoid this, **double buffering** should be used in animations.

You can achieve this using the command **enableRepaint(False)**, causing the graphics command to be executed only in a background buffer (off screen buffer) and effects are no longer visible. So *clear()* only deletes the background buffer and does not erase the graphics window anymore. You must trigger the display of the graphics buffers on the screen (called **rendering**) yourself at the right moment by calling **repaint()**.

In this program you also need to globally distinguish the variables *x* and *v* in the function *step()* since they are changed in the function.

EXERCISES

1. If you do not move the x- and y-coordinates with ordinary cosine or sine functions as you did in your first program, but rather at different rates, it will create interesting curve patterns called **Lissajoux** figures. Draw such figures with a resolution of 1/1000 in the range of t = 0 to 2π and with

   x = cos(4.5 * t) und y = sin(6.3 * t)

2. Instead of using fixed numbers, use the variables omega_x and omega_y to draw the figure for the following values:

omega_x	omega_y
3	5
3	7
5	7

Do you see a connection between the figure and the values of omega_x and omega_y?

3. Draw the Lissajoux figure with omega_x = 2 and omega_y = 7, in the range of t = 0 to 2π, and with a resolution of 1/100 in a GPanel with the coordinates -2 to 2 (both axes). Instead of connecting the points with lines, draw a circle with a radius 0.2 at any point. You get a "slinky-like" figure. As you can see in the picture, you can make monochrome circles or you can fill them with color using *getRandomX11Color()*. Play around with it!

3.6 KEYBOARD CONTROLS

INTRODUCTION

Programs become especially interactive when the user can control the program execution by using keys on the keyboard. Although keystrokes are actually events that always occur independently of the program, they can be captured also through querying functions.

PROGRAMMING CONCEPTS: *Boolean data type, game state, animation*

KEYBOARD CONTROLS

The command **getKeyCodeWait()** will stop the program until you press a key. Then, the function provides you with the corresponding key code as a return value. With the exception of certain special keys, each key has its own numerical code.

You can figure out the key codes using a simple test program. The numerical codes are **written** in the console window.

```
from gpanel import *
makeGPanel(0, 10, 0, 10)

text(1, 5, "Press any key.")
while True:
    key = getKeyCodeWait()
    print key,
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)


```
83 70 72 37 40 38 39
```

MEMO

You can use the command **getKeyCodeWait()** for keyboard inputs. The computer waits until you press a key and then returns the key code.

However, you have to remember that the GPanel window must be active. In other words, it must be in focus. If the window loses focus, you have to click somewhere inside it in order to activate it again. Only the currently active window receives keyboard events.

CONTROLLING FIGURES

You can move graphic objects using the keyboard. The program controls the green circle with the cursor keys, moving it left, right, up, or down. In a so-called **event loop** the program waits for a key press and then processes the obtained key code in a nested *if-else* structure.

Since the drawing of the circle is used over and over again, it makes sense that you would pack the code into its own function **drawCircle()** that can be called multiple times, in compliance with the structured programming paradigm.


```
from gpanel import *

KEY_LEFT = 37
KEY_RIGHT = 39
KEY_UP = 38
KEY_DOWN = 40

def drawCircle():
    move(x, y)
    setColor("green")
    fillCircle(5)
    setColor("black")
    circle(5)

makeGPanel(0, 100, 0, 100)
text("Move the circle with the arrow keys.")
x = 50
y = 50
step = 2
drawCircle()

while True:
    key = getKeyCodeWait()
    if key == KEY_LEFT:
        x -= step
        drawCircle()
    elif key == KEY_RIGHT:
        x += step
        drawCircle()
    elif key == KEY_UP:
        y += step
        drawCircle()
    elif key == KEY_DOWN:
        y -= step
        drawCircle()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

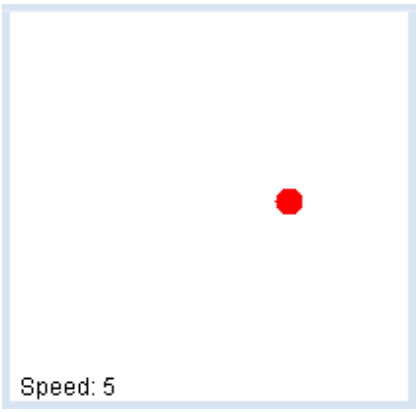
MEMO

To make the program more readable, you can introduce **constants** for the keyboard codes of the arrow keys. So that they are especially noticeable, constants should be placed in the program header and written in capital letters.

NON-BLOCKING KEYBOARD QUERIES

As you are probably aware, the keyboard is often used to control the game play in computer games. In this case you can of course not use the blocking function *getKeyCodeWait()* because it would pause the game. Rather, you need a function that will deliver the information if a key has been pressed, but that also immediately returns.

In case you indeed pressed a key, you process this event, otherwise the game will continue on normally.



You want the speed of a ball moving back and forth to get slower with the letter key 's' (for slow) and get faster with 'f' (for fast), but only up to a certain maximum value. You need to again focus your attention on the *event loop*, which is where everything essential happens. In it, *kbhit()* periodically queries, whether a key was hit or not. If this is the case, **kbhit()** returns True and you can get the key code by using **getKeyCode()**.

```
from gpanel import *
import time

KEY_S = 83
KEY_F = 70

makeGPanel(0, 600, 0, 600)
title("Key 'f': faster, 's': slower")

enableRepaint(False)
x = 300
y = 300
v = 10
vmax = 50
isAhead = True

while True:
    startTime = time.clock()

    if kbhit():
        key = getKeyCode()
        if key == KEY_F:
            if v < vmax:
              v += 1
        elif key == KEY_S:
            if v > 0:
              v -= 1

    clear()
    setColor("black")
    text(10, 10, "Speed: " + str(v))
    if isAhead:
        x = x + v
    else:
        x = x - v
    move(x, y)
    setColor("red")
    fillCircle(20)
    repaint()
    if x > 600 or x < 0:
        isAhead = not isAhead
    while (time.clock() - startTime)  < 0.010:
        delay(1)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

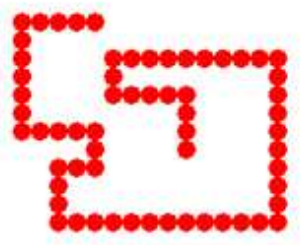
Since it is an animation, we again need to use an **animation timer** in order to obtain a run through the event loop that is as periodic as possible. The next game state is created in the loop and it is then displayed in the window screen with **repaint()**.

kbhit() returns a truth value, which we refer to as a **boolean**. If a key was pressed since the last call, it returns *True*, otherwise *False*.
In order to move the ball to the right (forward), its x-coordinate must increase by *v* (the measure for speed) with each pass of the event loop. To move to the left, the coordinate *v* must decrease. We summarize a forward and backward movement as a **game state** which we save in the variable *isAhead*.

You can add a word to a second word in *Python* using the + character, for example "Hans" + "Peter" results in the word "HansPeter". However, if you want to add a number to a word, you first have to convert the number using the *str()* function.

EXERCISES

1. Using the cursor keys UP, DOWN, LEFT and RIGHT draw a snake line of small red circles, which lie closely next to each other.



2. As an extension, define the following buttons for color selection: after you press the letter key *g*, the circles should turn green, *b* should make them blue, and *r* red.

3. Extend the above program with the ball moving back and forth so that the UP and DOWN cursor keys make the ball move up and down.

3.7 MOUSE EVENTS

INTRODUCTION

So far, your understanding of the computer is that it executes instruction after instruction. It can also change the course of a program due to certain conditions or run through loops. The corresponding program structures are called **sequence, selection,** and **iteration.** As early as 1966 Böhm and Jacopini proved in a famous **article** that any calculation procedures (algorithms) can be realized using these three programming structures.

This is, however, only true as long as you do not incorporate any external influences. For example, you can cancel a program at any moment by clicking with the mouse on "Close" (close button). Such processes need a new programming concept: **event control** (event handling). You have already learned the basic principles in the chapter Turtle Graphics/Event Control. It consists in procedures of the type:

"Whenever the event *e* occurs, the function *f* is executed".

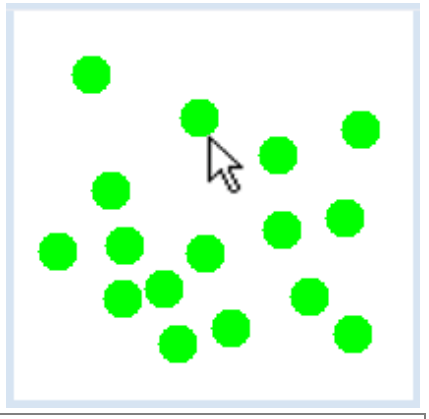
The implementation is simple and known since the early days of computer technology in the fifties of the last century. We define a function *f* (then called interrupt routine) which is never even called by our own program. It sleeps, so to speak, until a certain event *E* occurs, an external influence, upon which it is then automatically called by the system. Today we call such a function **callback** and we say that the **callback *f* is "fired" by the event *E***. Often, callbacks are called with parameter values that contain important information about the event, for example, which mouse button was pressed or where the mouse is located.

PROGRAMMING CONCEPTS: *Event-driven program, callback, registering callbacks*

REACTING TO A MOUSE EVENT

You can also use mouse events in the GPanel, just like in turtle graphics. In the first example, a green circle is drawn at the current mouse position when the left or right mouse button is pressed. Do the following:

First, in a function with an arbitrarily chosen name, define what should happen when a mouse button is pressed. Here you choose a name, for example **onMousePressed()**, that expresses what the function does as well as possible. When called by the system, the callback receives the current coordinates of the mouse cursor. Next you need to tell the system that it should call your callback when the mouse button is pressed. This process is called **callback registration**. To register your callback you will need a named parameter of *makeGPanel()* that is called **mousePressed**.



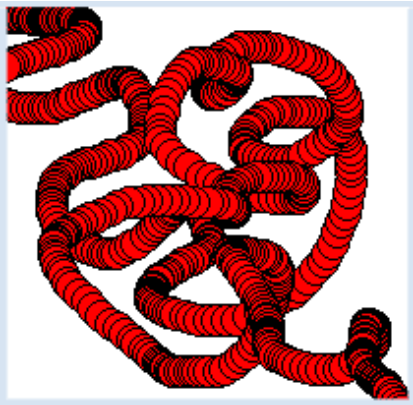
```
from gpanel import *

def onMousePressed(x, y):
    move(x, y)
    fillCircle(0.02)

makeGPanel(mousePressed = onMousePressed)
setColor("green")
```

MEMO

A callback is **not** called by your own program, but rather automatically when the event is triggered. The registration of the callback is performed through a named parameter. You can detect the pressing of a mouse button with two different callbacks: a click event or a press event. The click event will not be triggered until after the key is released, but the press event triggers immediately once you press the button.

DETECTING MOUSE MOVEMENT

The mouse movement can also be recognized as an event, which is triggered in rapid succession when the mouse is moved. The parameter is called **mouseMoved**. Your program draws a red filled circle with a black outline at every call of the **callback**, whereby you can draw fun tubelike pictures.



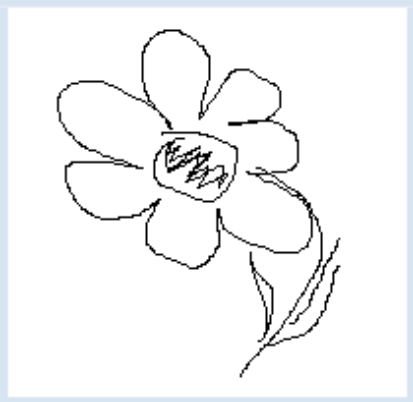
```
from gpanel import *

def onMouseMoved(x, y):
    move(x, y)
    setColor("red")
    fillCircle(.04)
    setColor("black")
    circle(.04)

makeGPanel(mouseMoved = onMouseMoved)
```

MEMO

The **onMouseMoved(x, y)** callback is registered through a named parameter *mouseMoved.*

FREE HAND DRAWING WITH A PRESSED MOUSE BUTTON

Now you are already capable of writing a simple drawing program, with which you can draw a figure free-handedly using the mouse. All you need is the drag event which is triggered in rapid succession when you move the mouse with the button **pressed down**. The program logic is simple: move the graphics cursor to the current location when the **press event** occurs and then draw a line using *draw()* in the **drag event** callback.



```
from gpanel import *

def onMousePressed(x, y):
    move(x, y)

def onMouseDragged(x, y):
    draw(x, y)

makeGPanel(mousePressed = onMousePressed,
           mouseDragged = onMouseDragged)
```

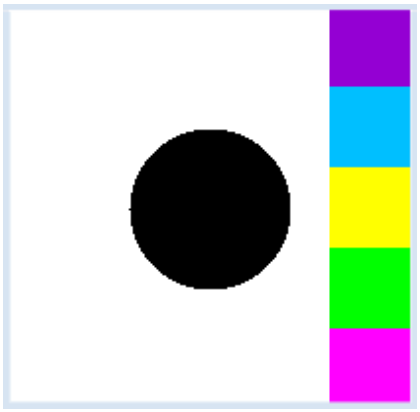
MEMO

You can register **multiple callback** with named parameters simultaneously. The order of the parameters does not matter.

THE LEFT AND RIGHT MOUSE BUTTON

As you have probably noticed, the mouse events are triggered by both the left and right mouse buttons. If you want to differentiate the two buttons, use the functions **isLeftMouseButton()** and **isRightMouseButton()**. These return *True* when the left or the right button is involved, respectively.

When you press on the **right mouse button** the program opens a color palette. You can then select the fill color of the circle with the **left mouse button**.



```
from gpanel import *

def onMousePressed(x, y):
  if isLeftMouseButton():
      pixColor = getPixelColor(x, y)
      if pixColor == makeColor("white"):
          return
      clear()
      setColor(pixColor)
      move(5, 5)
      fillCircle(2)

  if isRightMouseButton():
      for i in range(5):
          move(9, 2 * i + 1)
          if i == 0:
              setColor("deep pink")
          if i == 1:
              setColor("green")
          if i == 2:
              setColor("yellow")
          if i == 3:
              setColor("deep sky blue")
          if i == 4:
              setColor("dark violet")
          fillRectangle(2, 2)

makeGPanel(0, 10, 0, 10, mousePressed = onMousePressed)
move(5, 5)
fillCircle(2)
```

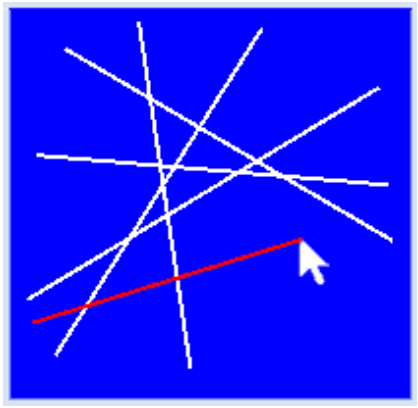
MEMO

The registered mouse callbacks are triggered with the left and the right mouse buttons. You can find out which button was used by calling **isLeftMouseButton()** or **isRightMouseButton()**.

RUBBER BAND LINES

If you want to draw lines with a drawing program, you can mark the starting point by pressing the mouse button. While dragging the mouse, you make a temporary line similar to that of a rubber band that is fastened at the starting point. Only release the mouse button when you are satisfied with the position of the line, and then it will actually be drawn.

So here you need three callbacks: **onMousePresssed**, **onMouseDragged** and **onMouseReleased**.



But there is a particular problem: to move the rubber band over the drawing area it must be repeatedly erased from its old location and drawn again to the new location, without changing the already existing drawing. If you deleted the lines by overwriting them with the background color, gaps would result in the existing drawing at the intersection points.

To solve this problem, you must save the existing drawing in the press callback (one also calls this "rescue"). The deletion of the temporary rubber band then happens by restoring this "old" drawing. You can save the drawing with **storeGraphics()** and restore it with **recallGraphics()**.

```
from gpanel import *

def onMousePressed(x, y):
    global x1, y1, x2, y2
    storeGraphics()
    x1 = x
    y1 = y
    x2 = x1
    y2 = y1
    setColor("red")

def onMouseDragged(x, y):
    global x2, y2
    recallGraphics()
    x2 = x
    y2 = y
    line(x1, y1, x2, y2)

def onMouseReleased(x, y):
    setColor("white")
    if not (x1 == x2 and y1 == y2):
        line(x1, y1, x2, y2)

x1 = 0
y1 = 0
x2 = 0
y2 = 0

makeGPanel(mousePressed = onMousePressed,
           mouseDragged = onMouseDragged,
           mouseReleased = onMouseReleased)
title("Press And Drag To Draw Lines")
bgColor("blue")
```

```
setColor("white")
lineWidth(2)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Remember the principles of drawing rubber band lines:

In a **press event** the end point of the line is initialized and the graphic is saved.

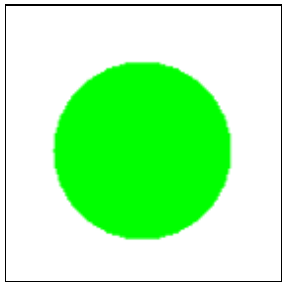
In a **drag event** the saved/stored graphic is restored, the temporary line with the new end point is saved, and the new end point is saved.

In a **release event** the line is definitely drawn, but only if the mouse was really moved.

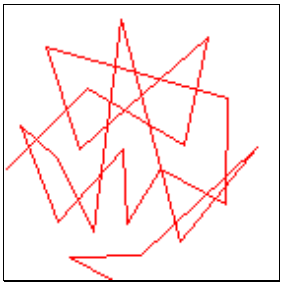
EXERCISES

1. Draw a green filled circle. The fill color should change to red when you move the mouse onto the circle. It should turn back to green when the mouse moves off.

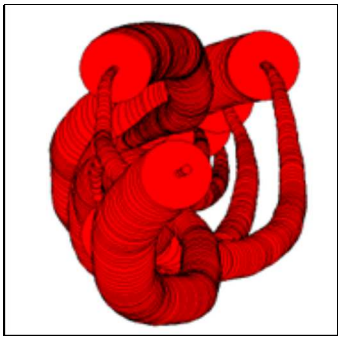
   You can specify a window with
   makeGPanel(-10, 10, -10, 10,
   mouseMoved = onMouseMoved)



2. Your program should draw a line segment after every mouse click.

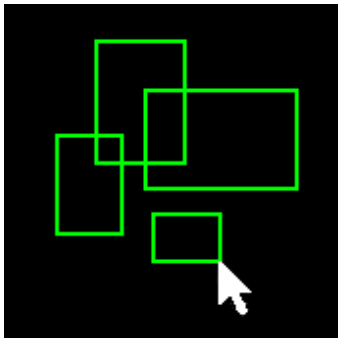


3. Upon the movement with a **pressed** mouse button, your program should draw a tube-like figure. While moving, the tube should swell up from an initial thickness of 0.01 to 0.1, and then return back to its original thickness.



4. Write a program where you can draw green rectangles onto a black background. In this case, you should be able to place a temporary "rubber band rectangle" by pressing

and dragging the mouse, before it is definitely placed upon releasing the mouse. Use the rectangle functions that are called with the coordinates of two opposite corner points of the rectangle (rectangle(x1, y1, x2, y2)).



ADDITIONAL MATERIAL

REGISTERING CALLBACKS WITH DECORATORS

Instead of using named parameters of *makeGPanel()* to register a callback, an arbitrary named function with two parameters x and y can be "decorated" by a preceding line, so that TigerJython automatically registers the function as callback that is called when the event happens. The additional line has to be prefixed by the "at" sign @. The following decorators are available:

@onMousePressed	mouse button is pressed
@onMouseReleased	mouse button is released
@onMouseClicked	mouse button is pressed and released
@onMouseDragged	mouse is moved while a button is pressed
@onMouseMoved	mouse is moved while no button is pressed
@onMouseEntered	mouse enters the graphics window
@onMouseExited	mouse leaves the graphics window

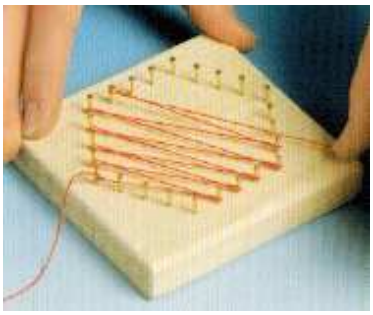
So the program shown above which draws a circle when the mouse button is pressed, can be written using a decorator:

```
from gpanel import *

@onMousePressed
def doIt(x, y):
    move(x, y)
    fillCircle(0.02)

makeGPanel()
setColor("green")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

3.8 THREAD GRAPHICS

INTRODUCTION

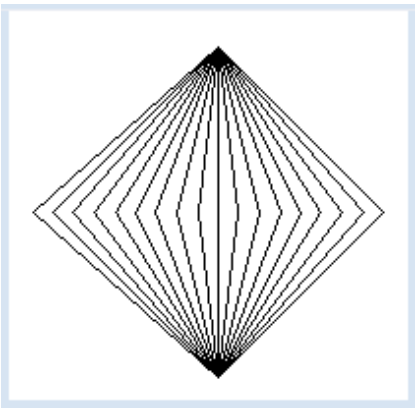

From Täubner, Walz: Fadengrafik

You have likely played with thread graphics already in preschool. For this, you hammered nails or inserted needles into a timber or carton board to create a particular figure, according to a crafting guide. Most of these were arranged at equal intervals and you linked them together with threads. When you placed a sufficient amount of threads, interesting curves appeared where the thread consolidated. In mathematics, this is called **envelope** (also envelope curve) because the threads are tangent to this curve.

Instead of creating the thread graphic yourself, you can also instruct a machine to do it. This would require the machine to not only understand the instructions but then to also translate these instructions into an action, for example using a robot arm to pull the strings or record the strings on a screen. Such an instruction manual for a machine is also called an **algorithm**. You can first formulate the algorithm as a "craft" instruction understandable in colloquial language. Since it is desirable that the machine produces the exact same pattern on each pass, the algorithm must be formulated so precisely that the machine knows exactly what to do at every step. Programming languages were invented for this and that is why you learn to program, since in the natural languages there is no such unambiguity.

PROGRAMMING CONCEPTS: *Algorithm, data structure, model, program elegance, list, index*

POINTS AS LISTS



Instead of working with boards, nails, and threads, you can transfer the procedure to your computer. Thereby you make an **portrayal of nature**, you **model** the board as a screen window, the nails as points on the screen, and the threads as lines.

In transferring the algorithm into a programming language, it is important to establish the closest relationship possible to reality. Nails, and geometric points respectively, represent tangible objects to you, and so they should be in the program as well.

In geometry, you can write P(x, y) for a point, where x and y are the coordinates. In the program, we can pack the two numbers x and y into a data structure, called a **list**. We write p = [x, y]. The geometric point P(0, 8) is thus modeled by the list **p = [0, 8]** .

You can access the individual components of a list with an **index** with a count starting at 0. You have to write the index in a set of square brackets, so **p[0]** for the x-coordinate, and p[1] for the y-coordinate. The nice thing is that all of the graphic functions of the GPanel are "list conscious" because they also work with point lists instead of x-y-coordinates. Your program models the pulling of threads from nail *A* around 19 nails at the coordinates on the x-axis to nail *B*, and back again. You can even incorporate a *delay()* which causes the stringing to take a longer time that is graspable by humans.

```
from gpanel import *

DELAY = 100

def step(x):
    p1 = [x, 0]
    draw(p1)
    delay(DELAY)
    draw(pB)
    delay(DELAY)
    p2 = [x + 1, 0]
    draw(p2)
    delay(DELAY)
    draw(pA)
    delay(DELAY)

makeGPanel(-10, 10, -10, 10)
pA = [0, 8]
pB = [0, -8]
move(pA)
for x in range(-9, 9, 2):
    step(x)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

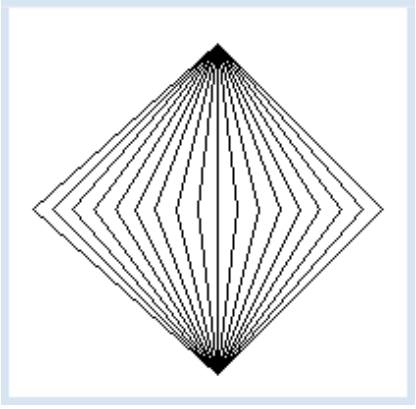
MEMO

The data must also be structured conveniently in the implementation of an algorithm. Our geometric points are modeled as a list with two elements (x- and y-coordinates). The choice of the **data structure** significantly affects the program. Niklaus Wirth, a famous computer science professor at the ETH Zürich, aptly said: **program = algorithm + data structure** [**Ref.**]
Lists can store multiple values, named list elements. They are defined with square brackets. You can read the individual elements with a list index and **assign new values**.

All of the graphics commands of GPanel also work with points modeled as lists of x- and y-coordinates.

PROGRAMMING IS AN ART

You probably realize that you can create the previous thread graphic much easier if you draw the lines independently of how the thread would actually be drawn by hand. You just need to connect the points A and B with routes.



```
from gpanel import *

makeGPanel(-10, 10, -10, 10)
pA = [0, 8]
pB = [0, -8]

for x in range(-9, 10, 1):
    pX = [x, 0]
    line(pA, pX)
    line(pB, pX)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

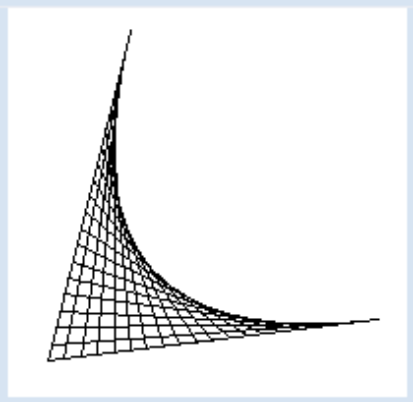
MEMO

An algorithm can be implemented in various ways that differ in length of code and duration of the execution of the program. We also speak of more elegant and less elegant programs. Just remember that it is not enough for a program to produce a correct result, but it should also be written **elegantly**. Consider programming an art!

ELEGANT THREAD GRAPHIC ALGORITHMS

You often need dividing points of a line segment for thread graphics. For this there is a simple function in GPanel called **getDividingPoint(pA, pB, r)**, which you pass the two endpoints pA and pB of the line and the division factor *r.* It returns you the dividing point as a list [**more...**].

You are now modeling a thread graphic with nails on the sides AB and AC with an especially elegant program.



```
from gpanel import *

makeGPanel(0, 100, 0, 100)

pA = [10, 10]
pB = [90, 20]
pC = [30, 90]

line(pA, pB)
line(pA, pC)

r = 0
while r <= 1:
    pX1 = getDividingPoint(pA, pB, r)
    pX2 = getDividingPoint(pA, pC, 1 - r)
    line(pX1, pX2)
    r += 0.05
    delay(300)
```

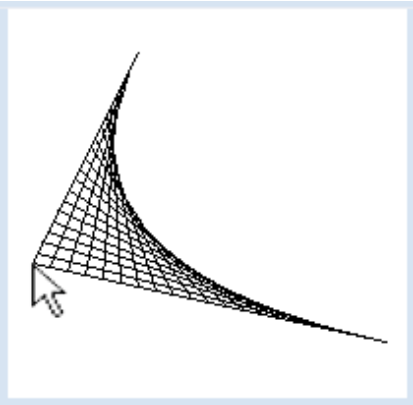
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Library functions such as *getDividingPoint()* can greatly simplify a program. For certain well-defined tasks, you should use existing library functions that you know from your programming experience here, taken from documentations, or from what you can find on the Web.

Mathematically, the resulting curve is a **quadratic Bézier curve**. You can draw it with the function *quadraticBezier(pB, pA, pC),* where pB and pC are the endpoints, and pA is the control point of the curve.

MOUSE CONTROLLED THREAD GRAPHICS



Modeling natural processes with the computer is not just a game, it also has versatile applications. You can test different situations in a much shorter time and with much less effort with a computer until you have found one that you want to implement into practice. Your program is particularly attractive if you can make changes with the mouse that have an immediate effect. With *Python,* this can be incorporated with little extra effort, by using callbacks.

In your program, you can move the vertex *A* by moving the mouse, and a new thread graphic will be made immediately.

In order to create the graphics, you use the function **updateGraphics()** which is called by the **mouse callbacks**. Every time you delete the entire graphics window and then recreate it with point *A* at the current location of the mouse cursor.

```
from gpanel import *

def updateGraphics():
    clear()
    line(pA, pB)
    line(pA, pC)
    r = 0
    while r <= 1:
        pX1 = getDividingPoint(pA, pB, r)
        pX2 = getDividingPoint(pA, pC, 1 - r)
        line(pX1, pX2)
        r += 0.05

def myCallback(e):
    pA[0] = toWindowX(e.getX())
    pA[1] = toWindowY(e.getY())
    updateGraphics()

makeGPanel(0, 100, 0, 100,
              mousePressed = myCallback,
              mouseDragged = myCallback)

pA = [10, 10]
pB = [90, 20]
pC = [30, 90]
updateGraphics()
```

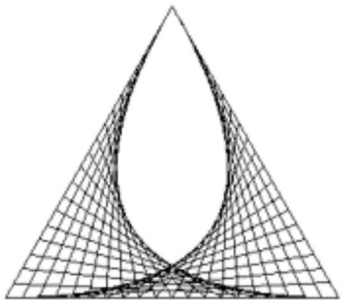
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

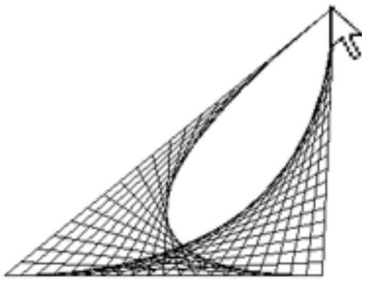
You can also deal with two different events, here the press event and the drag event, using the same callback.

EXERCISES

1. Create the adjacent thread graphic

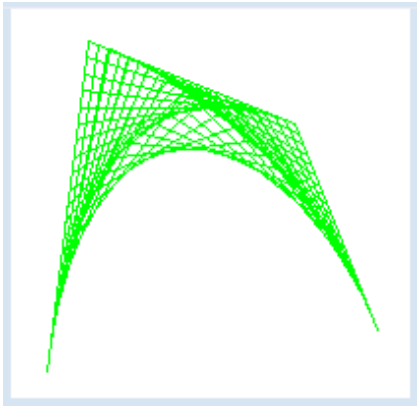


2. Work from the thread graphic from exercise 1, so that you can draw the top of the triangle with a mouse drag and the graphic is repeatedly drawn anew.



EXTRA MATERIAL

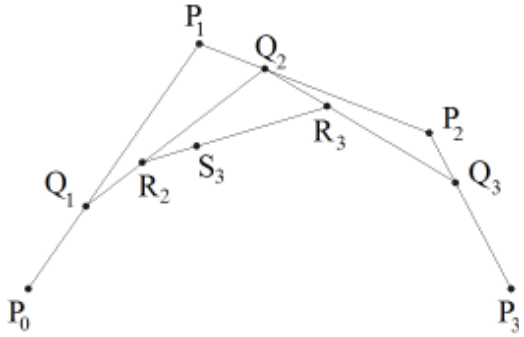
BÉZIER CURVES



These curves were invented in the sixties of the last century by Pierre Bézier, then an engineer of the car company Renault, so one could produce aesthetically pleasing curves for the design of industrial products.

You can create a cubic Bézier curve as a thread graphic using the De Casteljau algorithm.

The algorithm reads as follows:



- Specify 4 points P0, P1, P2, P3. (P0 and P3 will be the end points of the curve, P1 and P2 so-called control points)
- Connect P0P1, P1P2, P2P3
- The routes P0P1, P1P2, P2P3 are arranged into equidistant division points. For a given division ratio, this results in the dividing points Q1, Q2, Q3
- Connect Q1Q2, Q2Q3
- Split the routes Q1Q2, Q2Q3 in the same division/factor ratio. This results in the division points R2 and R3

▷ Connect R2R3

You can easily implement the algorithm into a program if you implement the points as lists and call the function **getDividingPoint()** several times.

```
from gpanel import *

makeGPanel(0, 100, 0, 100)

pt1 = [10, 10]
pc1 = [20, 90]
pc2 = [70, 70]
pt2 = [90, 20]

setColor("green")

line(pt1, pc1)
line(pt2, pc2)
line(pc1, pc2)

r = 0
while r <= 1:
    q1 = getDividingPoint(pt1, pc1, r)
    q2 = getDividingPoint(pc1, pc2, r)
    q3 = getDividingPoint(pc2, pt2, r)
    line(q1, q2)
    line(q2, q3)
    r2 = getDividingPoint(q1, q2, r)
    r3 = getDividingPoint(q2, q3, r)
    line(r2, r3)
    r += 0.05

setColor("black")
#cubicBezier(pt1, pc1, pc2, pt2)
```

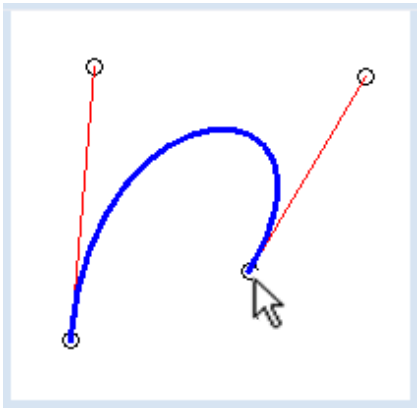
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

A cubic Bézier curve is defined by 4 points. You can draw one in GPanel with the function *cubicBezier()*. The current drawing color and line thickness will be used.

INTERACTIVE CURVE DESIGN

Combining your knowledge, you can already write a quite professional program with which you can create a Bézier curve and interactively change it with the mouse. The program even notices when you are near one of the 4 points with the cursor and colors it. With a press event you can then grab the point and move it around.



The four points must be run through multiple times in the program. It is therefore advisable that you also put them in a list with the name **points**, so that you can process them with a *for* structure.

It is also important that you know which of the points you have just grabbed. You store this information in the variable **active**: if none of the points are grabbed it has the value -1, otherwise its value corresponds to the index of the corresponding point.

```
from gpanel import *

def updateGraphics():
    # erase all
    clear()

    # draw points
    lineWidth(1)
    for i in range(4):
        move(points[i])
        if active == i:
            setColor("green")
            fillCircle(2)
        setColor("black")
        circle(2)

    # draw tangents
    setColor("red")
    line(points[0], points[1])
    line(points[3], points[2])

    # draw Bezier curve
    setColor("blue")
    lineWidth(3)
    cubicBezier(points[0], points[1], points[2], points[3])

def onMouseDragged(e):
    if active == -1:
        return
    points[active][0] = toWindowX(e.getX())
    points[active][1] = toWindowY(e.getY())
    updateGraphics()

def onMouseReleased(e):
    active = -1
    updateGraphics()

def onMouseMoved(e):
    global active
    x = toWindowX(e.getX())
    y = toWindowY(e.getY())
    active = near(x, y)
    updateGraphics()

def near(x, y):
    for i in range(4):
        rsquare = (x - points[i][0]) * (x - points[i][0]) +
                  (y - points[i][1]) * (y - points[i][1])
        if rsquare < 4:
            return i
    return -1

pt1 = [20, 20]
pc1 = [10, 80]
pc2 = [90, 80]
pt2 = [80, 20]
points = [pt1, pc1, pc2, pt2]
active = -1

makeGPanel(0, 100, 0, 100,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased,
    mouseMoved = onMouseMoved)
updateGraphics()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

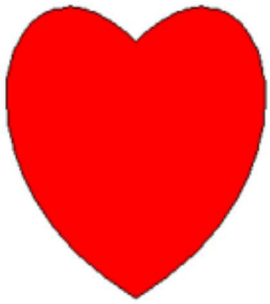
MEMO

There are also complicated data structures such as lists whose elements are again lists. For example, you can address the x-coordinate of P1 using the **points[1][0]**, thus with **double brackets**.

Today, Bézier curves are important design tools in the CAD domain [**Ref**.]

EXERCISES

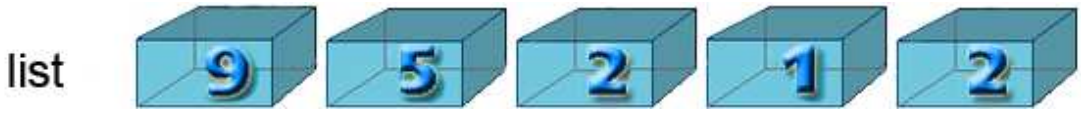
1. The heart consists of two cubic Bézier curves with the same start and end points, plus symmetrical control points. On a piece of paper, draw a sketch of where these points should be placed and then create the drawing. The filling is made with the function *fill(point, old_color, new_color),* where *point* stands for an inner point of a bordered area.



3.9 Lists

INTRODUCTION

Sometimes you have to store values that belong together, but their exact number is not known during the creation of the program. Because of this, you will need a data structure where you can store multiple values. The structure should be flexible enough to take the **order** of the added values into account. It is obvious in this case to use a sequence of simple containers, which you have already heard about, namely a **list**. Here you will find out in detail how to work with lists.



A list with 5 elements

A list consists of individual elements arranged one after the other. In contrast to an unstructured set of elements, there is a **first** and a **last** element, and all the other elements have a **predecessor** and a **successor**.

Lists (and similar containers) are enormously important for programming. The operations possible with lists are very descriptive. The most important are:

- Adding elements (at the end, at the beginning, somewhere in between)
- Reading elements
- Changing elements
- Removing elements
- Iterating all elements
- Sorting elements
- Searching for elements

In Python you can store any data in lists, not only numbers. The individual elements can even have a different type and you can, for example, store numbers and letters in the same list.

PROGRAMMING CONCEPTS: *Container, list, predecessor, successor, reference variable*

GRADE LIST

You can interpret a list as a variable. It thus has a name and a value, namely its **elements**. You create it with a pair of square brackets, e.g. *list* = [1, 2, 3] generates a list with the elements 1, 2 and 3. A list can also be empty. You can define an empty list with *list = []*.

Grade books, where you enter the grades for a particular school subject, are a typical use of lists, let's say biology grades. At the beginning of the semester you have an empty list, which is expressed in Python as **bioGrades = []**. Writing in grades is then equivalent to adding list items. In Python you use the command *append()*, so for a score of 5 it looks like this: **bioGrades.append(5)**.

You can view the list at any time with a print command, just simply write *print bioGrades*.
If you want to calculate your grade point average, you have to **run through** the list. You can do this easily and elegantly with a *for* loop, because

for grade in bioGrades:

copies every list value in order to the variable *grade,* and you can then use this in the loop body.

```
bioGrades = []
bioGrades.append(5.5)
print bioGrades
bioGrades.append(5)
print bioGrades
bioGrades.append(5.5)
print bioGrades
bioGrades.append(6)
print bioGrades
sum = 0
for note in bioGrades:
    sum += note
print "Average: " + str(sum / len(bioGrades))
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Using the method *append()* you can add new elements to the end of the list.

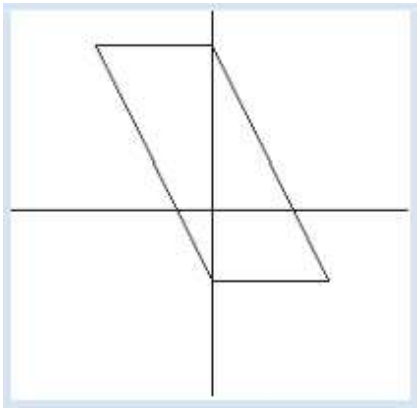
The built-in function **len()** returns the current length of the list. Note the interesting trick with the variable **sum**, with which you can create the sum to then calculate the average. You can also obtain the sum directly with the built-in function *sum(bioGrades)*.

LIST WITH A FIXED NUMBER OF ELEMENTS

It is often already known how long a container list has to be, and that all elements have the same data type, when creating the program. In many programming languages you call such a data structure an **array**. The individual elements are usually accessed via their index. In Python there is no such data type and instead you use a list.

The program defines a polygon as a list with 4 vertices (these are again defined as lists with 2 coordinates). In order to access them with indices from the start, create a list with 4 zeros *polygon = [0, 0, 0, 0].* You can also use the shorthand notation *polygon = [0] * 4*.

After that, you copy in the 4 vertices, which replaces the zeros by point lists. With a *for* loop you display the polygon.



```
from gpanel import *

pA = [0, -3]
pB = [5, -3]
pC = [0, 7]
pD = [-5, 7]

makeGPanel(-10, 10, -10, 10)
line(-10, 0, 10, 0)
line(0, -10, 0, 10)

polygon = [0] * 4  # list with 4 elements, initialized with 0
polygon[0] = pA
polygon[1] = pB
polygon[2] = pC
polygon[3] = pD
```

```
for i in range(4):
    k = i + 1
    if k == 4:
        k = 0
    line(polygon[i], polygon[k])
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

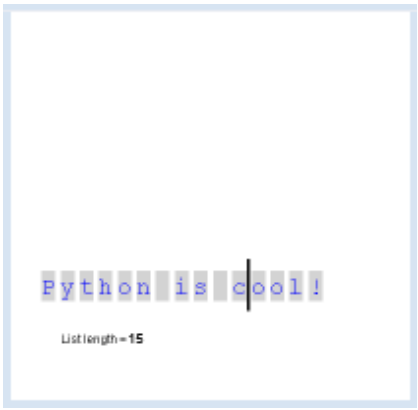
If you already know the length of the list when creating the program, generate a list with the initialization values 0 and then refer to the elements using the index.

INSERTING AND DELETING ELEMENTS

The program shows how a word processor works. The entered characters are inserted into a list of letters. It is clear that you do not know how many letters you will enter in the beginning, so a list is the ideal data structure. In addition, you see a text cursor which can be set to any position in the text with a mouse click.

When you type using a character key the letter is inserted to the right of the cursor and the list grows. When you use the backspace key the character to the left of the cursor is deleted and the list shrinks.

In order to represent everything nicely, you write the characters as text with a text color and background color in a GPanel. For this you run through the list with a list index *i*.




```
from gpanel import *

BS = 8
SPACE = 32
DEL = 127

def showInfo(key):
    text = "List length = " + str(len(letterList))
    if key != "":
        text += ". Last key code = " + str(ord(key))
    setStatusText(text)

def updateGraphics():
    clear()
    for i in range(len(letterList)):
        text(i, 2, letterList[i], Font("Courier", Font.PLAIN, 24),
             "blue", "light gray")
    line(cursorPos - 0.2, 1.7, cursorPos - 0.2, 2.7)

def onMousePressed(x, y):

    setCursor(x)
    updateGraphics()

def setCursor(x):
    global cursorPos
```

```
    pos = int(x + 0.7)
    if pos <= len(letterList):
       cursorPos = pos

makeGPanel(-1, 30, 0, 12, mousePressed = onMousePressed)

letterList = []
cursorPos = 0
addStatusBar(30)
setStatusText("Enter Text. Backspace to delete. Mouse to set cursor.")
lineWidth(3)

while True:
    delay(10)
    key = getKey()
    if key == "":
        continue
    keyCode = ord(key)
    if keyCode == BS:  # backspace
        if cursorPos > 0:
            cursorPos -= 1
            letterList.pop(cursorPos)
    elif keyCode >= SPACE and keyCode != DEL:
        letterList.insert(cursorPos, key)
        cursorPos += 1
    updateGraphics()
    showInfo(key)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You have already learned that you can access individual elements of a list using a list index that starts at zero. For this, you use the square brackets, i.e. **letterList[i]**. The index must always lay in the range of 0 and list length - 1. When you use a **for in range()** structure the stop value is just the length of the list.

You should never access an element that does not exist with the index. Errors with invalid index lists are one of the most common errors in programming. If you do not pay attention, you get programs that sometimes work and sometimes die.

To test which key was pressed you can use **getKey()**. This function returns immediately after the call and delivers either the character of the last key pressed or the value 65535 (the largest representable integer with 16 bit) if no key has been pressed.

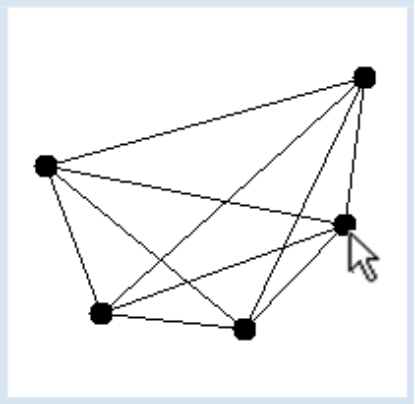
ALREADY A PROFESSIONAL PROGRAM

You are already able to visualize a graph with your knowledge [**more...**]

The task (also called "program specification") is the following:
You can create filled circles in the graphics window with a **right mouse click**, which are considered to be nodes of a graph, where the nodes are interconnected with named lines, called edges. Go on a node with the mouse; with a **left mouse press** and subsequent **dragging** you can move it around while the graph is updated constantly. If you right click on an existing node, it will be removed.

It is wise that you solve complex tasks by first looking at a **subtask** that has not yet met all of the final program specifications. For example, first write a program with which you can create nodes with each click. They should already be connected with all other existing nodes, but you cannot move them yet.

It seems obvious to model the graph with a **list graph** in which you store the node points.



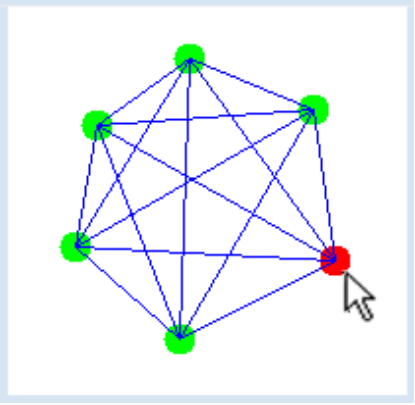
The nodes themselves are points with two coordinates P(x, y) that you model with a point list **pointlist [x, y]**. It is therefore a list, that then again contains lists as elements (but with a fixed length of 2). You accomplish joining the nodes with **double for loop**, but you must make sure that the nodes are only connected once.

```
from gpanel import *

def drawGraph():
    clear()
    for pt in graph:
        move(pt)
        fillCircle(2)

    for i in range(len(graph)):
        for k in range(i, len(graph)):
            line(graph[i], graph[k])

def onMousePressed(x, y):

    pt = [x, y]
    graph.append(pt)
    drawGraph()

graph = []
makeGPanel(0, 100, 0, 100, mousePressed = onMousePressed)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

Next you are going to incorporate the dragging and deletion of nodes. For this, you will need the right mouse button. As you drag it is important to know which node is being pulled. You can remember it by its index **iNode** in the *graph* list. If no node is being pulled, **iNode = -1**. In the function **near(x, y)**, using the Pythagorean theorem, you calculate the distance between point P(x,y) and all other points. Once one of the squared distances is less than 10, you abort the calculation and return the index of the node. Here you see that you can leave a function using *return* even in the middle of the procedure.

Everything else is fun programming work that you could also achieve yourself based on your current knowledge.



```
from gpanel import *

def drawGraph():
    clear()
```

```
    for i in range(len(graph)):
        move(graph[i])
        if i == iNode:
            setColor("red")
        else:
            setColor("green")
        fillCircle(2)

    setColor("blue")
    for i in range(len(graph)):
        for k in range(i, len(graph)):
            line(graph[i], graph[k])

def onMousePressed(x, y):
    global iNode
    if isLeftMouseButton():
        iNode = near(x, y)
    if isRightMouseButton():
        index = near(x, y)
        if index != -1:
            graph.pop(index)
            iNode = -1
        else:
            pt = [x, y]
            graph.append(pt)
    drawGraph()

def onMouseDragged(x, y):
    if isLeftMouseButton():
        if iNode == -1:
            return
        graph[iNode] = [x, y]
        drawGraph()

def onMouseReleased(x, y):
    global iNode
    if isLeftMouseButton():
        iNode = -1
        drawGraph()

def near(x, y):
    for i in range(len(graph)):
        p = graph[i]
        d = (p[0] - x) * (p[0] - x) + (p[1] - y) * (p[1] - y)
        if  d < 10:
            return i
    return -1

graph = []
iNode = -1
makeGPanel(0, 100, 0, 100,
              mousePressed = onMousePressed,
              mouseDragged = onMouseDragged,
              mouseReleased = onMouseReleased)
addStatusBar(20)
setStatusText("Right mouse button to set nodes, left button to drag")
```

MEMO

The program is fully **event-driven**. The main block only defines two global variables and initializes the graphics window. For each action the entire graphics window is cleared and rebuilt with the current situation of the graph.

The most important operations with lists:

li = [1, 2, 3, 4]	Defines a list with the numbers 1, 2, 3, 4
li = [1, "a", [7 , 5]]	Defines a list with different data types
li[i]	Accesses list elements with index i
li[start:end]	Sublist with elements from start to end, without end
li[start:end:step]	Sublist with elements from start to end, with the given step
li[start:]	Sublist with all elements starting at start
li[:end]	Sublist from the first element up to end, but without end
li.append(element)	Appends element at the end
li.insert(i, element)	Inserts element at position *i* (element *i* slides to the right)
li.extend(li2)	Appends all elements of li2 (concatenation)
li.index(element)	Finds the first occurrence and returns its index
li.pop(i)	Removes and returns the element with index *i*
pop()	Removes and returns the last element
li1 + li2	Returns the concatenation of li1 and li2 in a new list
li1 += li2	Replaces li1 by the concatenation of li1 and li2
li * 4	New list with elements of li repeated four times
[0] * 4	Makes a new list with length of 4 (all elements number 0)
len(li)	Returns the number of list elements
del li[i]	Removes the element with index i
del li[start:end]	Removes all elements from start to end, but without end
del li[:]	Removes the element with index i
li.reverse()	Reverses the list (last element becomes the first)
li.sort()	Sorts the list (comparison with standard methods)
x in li	Returns True, if x is (included) in the list
x not in li	Returns True, if x is not in the list

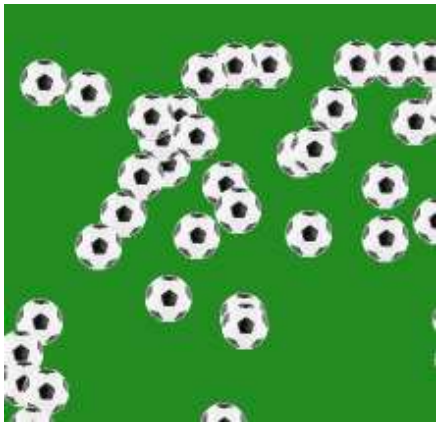
The notation with square brackets is called a **slice operation**. *start* and *end* are indices of the list. The slice operation works similarly to the parameters of *range()*.

EXERCISES

1. Input any number of grades with *inputFloat("prompt", False)*. If you press the *Cancel button*, the average will be written out in the console. Note that you must use the parameter value *False* so that the program does not end if you click *Cancel*. The special value *None* is returned at the termination, which as usual, you can test with *if*.

2. Extend the editor program above using the slice notation so that every right mouse click cuts away the beginning of the sentence, up to and including the first blank space.

3. Each time you click, a new image of a football (*football.gif*) should appear at the location of the mouse cursor. All of the footballs are constantly moving back and forth on the screen. Familiarize yourself with the football example in the chapter *animations*. You can optimize the program by loading the football image once with *img = getImage("sprites/football.gif")* and passing *img* to the function *image()*.



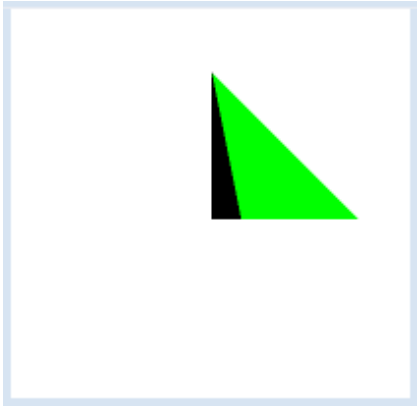
EXTRA MATERIAL:

MUTABLE AND IMMUTABLE DATA TYPES

In Python all data types are stored as objects, including the numeric types (integer, float, long, complex). As you know, you can access an object by its name. It is also said that the name **refers** to or is **bound** to the object. Therefore, such a variable is also called a **reference variable**.

A particular object can be referred to by more than one name. AN additional name is also called an **alias**. The following example shows how to deal with this.

A triangle is defined by the three vertex lists a, b, c. You can create an alias with the statement *a_alias = a*, so that *a* and *a_alias* both refer to the same list. If you alter the vertex list with the name *a*, the changes are obviously also visible in *a_alias*, since *a* and *a_alias* refer to the same list.

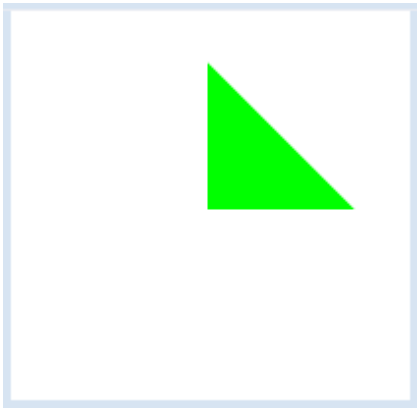


```
from gpanel import *

makeGPanel(-10, 10, -10, 10)

a = [0, 0]
a_alias = a
b = [0, 5]
c = [5, 0]

fillTriangle(a, b, c)
```

```
a[0] = 1
setColor("green")
fillTriangle(a_alias, b, c)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

Since numbers are also objects, you would expect the same behavior if you used numbers as vertex coordinate. However, the following example shows a different behavior. If you change *xA*, the value of *xA_alias* does not change.



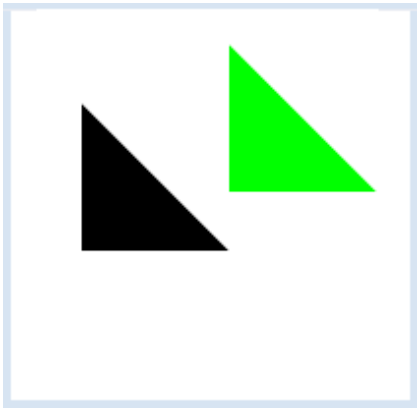
```
from gpanel import *

makeGPanel(-10, 10, -10, 10)

xA = 0
yA = 0
xA_alias = xA
yA_alias = yA
xB = 0
yB = 5
xC = 5
yC = 0

fillTriangle(xA, yA, xB, yB, xC, yC)
xA = 1
setColor("green")
fillTriangle(xA_alias, yA_alias, xB, yB, xC, yC)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

What is the explanation for that? The reason is that numbers are **immutable objects** and the statement *xA = 1* **generates a new number.** *xA_alias* is therefore still 0.

The difference between immutable and mutable data types can also be seen when passing parameters to functions. When an immutable object is passed, it cannot be changed inside the function. When a mutable object is delivered, the function can change the object. Such a change is called a **secondary** or **side effect**. In order to have a good programming style, you should use side effects sparingly because they can cause some annoying misconduct that is difficult to trace.

In the following example, the function *translate()* changes the passed vertex lists.



```
from gpanel import *

def translate(pA, pB, pC):
  pA[0] = pA[0] + 5
  pA[1] = pA[1] + 2
  pB[0] = pB[0] + 5
  pB[1] = pB[1] + 2
  pC[0] = pC[0] + 5
  pC[1] = pC[1] + 2

makeGPanel(-10, 10, -10, 10)

a = [0, 0]
b = [0, 5]
c = [5, 0]
fillTriangle(a, b, c)
translate(a, b, c)
setColor("green")
fillTriangle(a, b, c)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In Python all data are stored as objects, but some objects are considered to be **immutable**. These objects are: numerical data types, string, byte, and tuple.

All other data types are **mutable**. When you assign a new value to a variable of an immutable data type, a new object is created.

If mutable objects are passed to a function, the function can change the objects, while immutable objects are protected from such changes.

3.10 RANDOMNESS

INTRODUCTION

Chance plays an important role in your daily life. We can think of it as events that are not predictable. If you are asked to choose from the colors red, green, and blue, no one can predict which one you will choose and therefore the color is random. Chance plays a big role in games as well: If you roll a dice, the number of pips you get, between 1 and 6, is random.

Although the world is ruled by chance it is not chaotic, since even in chance there are regularities that allow for certain predictions. However, these only apply "on average", or in other words, if you are in the same situation many times. In order to investigate the laws of chance, you must make **random experiments** where you define the specific initial conditions, but where the process is controlled by random numbers.

The computer is exceptionally well suited for random experiments because it is easy to perform a large number of experiments. For this, the computer must generate a series of random numbers that are independent of each other. You most often use integers with a certain predetermined range, e.g. between 1 and 6, or a decimal number between 0 and 1. An algorithm that computes a set of random numbers is called a **random number generator**. It is important that the numbers occur with the same frequency as you would expect from a non-marked dice. We call such numbers **uniformly distributed** .

PROGRAMMING CONCEPTS: *Random numbers, random experiments, frequency, probability*

RANDOM PAINTING

You blot 20 colored ellipses with random sizes, random positions, and random colors onto a canvas. Whether you want to see this as a painting, or even as an artwork is up to you. Anyway, the resulting figures are fun. To determine the position and size of the ellipses, you can use the method **random()** from the **random module**, and a new random number will be delivered between 0 and 1 on every call. In order to obtain the random colors, you need three **random numbers between 0 and 255** that define the proportions of red, green, and blue color.

```
from gpanel import *
import random

def randomColor():
    r = random.randint(0, 255)
    g = random.randint(0, 255)
    b = random.randint(0, 255)
    return makeColor(r, g, b)

makeGPanel()
bgColor(randomColor())

for i in range(20):
    setColor(randomColor())
```

```
    move(random.random(), random.random())
    a = random.random() / 2
    b = random.random() / 2
    fillEllipse(a, b)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

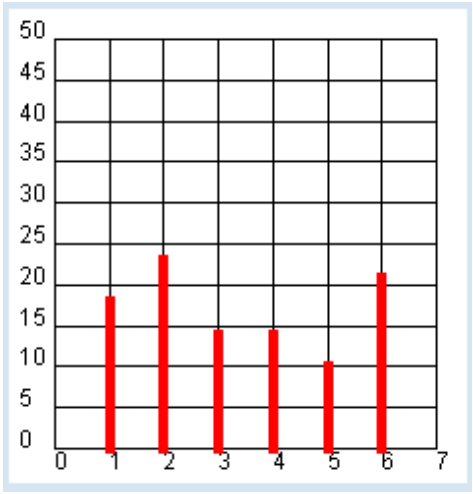
MEMO

random.random() returns uniformly distributed random numbers as floats between 0 (included) and 1 (excluded). You have to **import** the random module in order to access it. Colors are defined by their red, green, and blue parts (RGB). The values are integers between 0 and 255.

Using **randint(start, end)** you get a random integer between *start* and *end* (both included). The function *makeColor()* returns a colored object from 3 color values for red, green, and blue.

FREQUENCY OF DICE NUMBERS

One random experiment is to roll a dice 100 times to find out how often the numbers 1, 2,...6 occur.



You can run the experiment a lot faster with a computer. Instead of the dice, use **random numbers from 1 to 6**. You can show the frequency distribution graphically in a GPanel.

```
from gpanel import *
import random

NB_ROLLS = 100

makeGPanel(-1, 8, -0.1 * NB_ROLLS / 2, 1.1 * NB_ROLLS / 2)
title("# Rolls: " + str(NB_ROLLS))
drawGrid(0, 7, 0, NB_ROLLS // 2, 7, 10)
setColor("red")

histo = [0, 0, 0, 0, 0, 0, 0]
# hist = [0] * 7  # short form

for i in range(NB_ROLLS):
    pip = random.randint(1, 6)
    histo[pip] += 1

lineWidth(5)
for n in range(1, 7):
    line(n, 0, n, histo[n])
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The frequency of how often the individual pips occur must be saved. For this, you use the list **histo**, in which you add up the events at their corresponding index. You need a list with 7 elements because the index runs from 1 to 6.

Through some experiments, you can determine how the frequency of throwing increases the chance that the numbers **NB_ROLLS** are better balanced, and how they get increasingly close to 1/6 of the number of throws. This fact can be expressed as follows: **The probability to get one of the number of pips in dice throwing is 1/6**.

For the coordinate grid, call on *drawGrid(xmin, xmax, ymin, ymax, xticks, yticks)* with 6 parameters. The last two parameters determine the number of subdivisions. If *xmax* or *ymax* is a float, the axis labels will also be floats, otherwise they are integers.

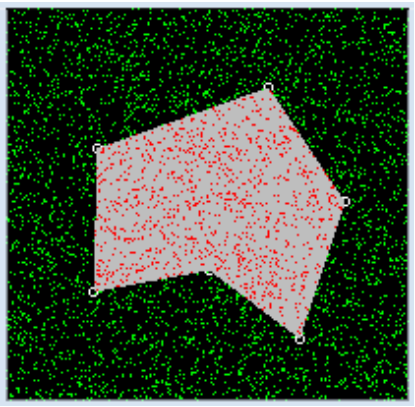
MONTE CARLO SIMULATION

The Principality of Monaco is world famous for its casino in the Monte Carlo district. The casino has not only been an attraction for celebrities for the past 150 years, but also for mathematicians who try to analyze the games and develop winning strategies. The computer is much better for testing these strategies and is actually better than the real game, because you do not loose any money with computer experiments as you do in the real games.

In the following "game", you throw points on a square area where there is a polygon. As an illustration, you can see the points as raindrops. As usual when it rains, there are always roughly about the same amount of drops in each unit area. So, the drops are **uniformly distributed**. You let a certain number of raindrops fall and then count how many of them fall onto the area of the polygon. It is obvious that the number will increase with an increasing surface area of the polygon, and that on average, it will be proportional to the surface area. For example, if you let drops fall onto a polygon with a surface area ¼ the size of the area of the surrounding square it will likely collect (on average) ¼ of all the raindrops. Once you have realized this, you can conversely find out the area by counting the number of the drops. Isn't this convenient?

The program is designed to be modern and user-friendly. With a **left mouse click** you can create the vertices of the polygon. You can then click with the **right mouse button** in the area that you would like to calculate, so that the polygon will be drawn and it will start to rain.

The result is displayed in the title bar.



```
from gpanel import *
import random

NB_DROPS = 10000

def onMousePressed(x, y):
    if isLeftMouseButton():
        pt = [x, y]
        move(pt)
        circle(0.5)
        corners.append(pt)
    if isRightMouseButton():
```

```
        wakeUp()

def go():
    global nbHit
    setColor("gray")
    fillPolygon(corners)
    title("Working. Please wait...")
    for i in range(NB_DROPS):
        pt = [100 * random.random(), 100 * random.random()]
        color = getPixelColorStr(pt)
        if color == "black":
            setColor("green")
            point(pt)
        if color == "gray" or color == "red":
            nbHit += 1
            setColor("red")
            point(pt)
    title("All done. #hits: " + str(nbHit) + " of " + str(NB_DROPS))

makeGPanel(0, 100, 0, 100, mousePressed = onMousePressed)
title("Select corners with left button. Start dropping with right button")
bgColor("black")
setColor("white")
corners = []
nbHit = 0
putSleep()
go()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

When you click with the **left mouse button** you are saving the vertices of the polygon into a list *corners* and drawing small circles as marks.

The actual rain simulation is performed in the function **go()**. It begins when you click the right mouse button and lasts for a certain amount of time. You make the falling raindrops visible with different colored points. If you directly call *go()* in the *pressCallback()*, as it might seem straightforward, you will see nothing until the simulation ends. The reason is that the system prevents refreshing the graphics in a mouse callback for system-intrinsic reasons. So if you want to visualize a longer-lasting action in a callback, it must happen in another part of the program. Often the main block of the program is used for this purpose. The execution is temporarily halted with **putSleep().** The press event awakens the sleeping main program with **wakeUp()** and the simulation will be carried out with the call *go()*.
In order to avoid problems in the future, you should always remember the following principle:

Callbacks must always return quickly. Therefore, no long-lasting actions should be executed there.

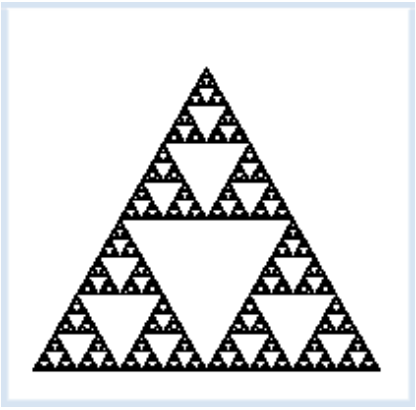
To find out if a raindrop has fallen onto the gray colored polygon area, use the following trick: You get the color of the point of impact with **getPixelColorStr()**. If it is the color gray (or red if another drop has already fallen there), you increase **nbHit by 1** and color the point red. You can test the procedure by generating some simple polygons (e.g. rectangles, triangles) and then by measuring the screen with a ruler. You will then realize that you need a lot of raindrops in order to obtain a reasonably accurate result [**more...**].

CHAOS GAME

It might at first seem surprising that you can create regular patterns with random experiments. This has to do with the compensation of statistical fluctuations for large numbers. In 1988 Michael Barnsley invented the following algorithm based on Chaos theory, which builds on a random selection of the vertices of a triangle:

1. Construct an equilateral triangle with the vertices A, B, C
2. Choose a point P in the interior
3. Randomly select one of the vertices
4. Halve the line segment from P to the vertex. This results in the new point P
5. Draw the point P
6. Repeat steps 2, 3, 4, 5

Such a formulation is common colloquially, but it cannot be directly translated into program code since step 6 requires that you should jump to step 3 again. In many modern programming languages, including *Python*, there is no jumping structure (no **goto**). Jumps must be implemented with one of the looping structures [**more...**].



```
from gpanel import *
import random

MAXITERATIONS = 100000
makeGPanel(0, 10, 0, 10)

pA = [1, 1]
pB = [9, 1]
pC = [5, 8]

triangle(pA, pB, pC)
corners = [pA, pB, pC]
pt = [2, 2]

title("Working...")
for iter in range(MAXITERATIONS):
    i = random.randint(0, 2)
    pRand = corners[i]
    pt = getDividingPoint(pRand, pt, 0.5)
    point(pt)
title("Working...Done")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

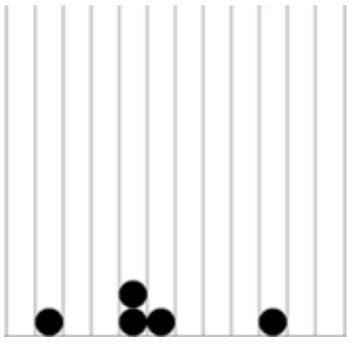
If you need a random object, you can join all of the objects in a **list** and pick an object out of it at a random index.

It is quite amazing that you can create a regular figure (called **Sierpinski triangle**) with randomly selected points.

EXERCISES

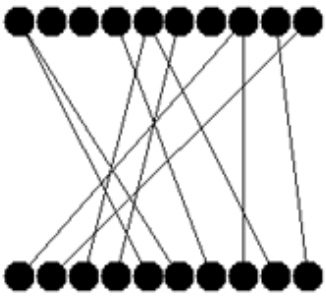
1. 5 kids meet at the playground and ask each other what month their birthday is in. It is quite surprising that the probability that at least two of them have the same birthday month is relatively large.

   Create a simulation with 100 random tests to determine this probability experimentally. Illustrate this by showing for each attempt of the experiment twelve rectangular containers in a GPanel, each of them standing for one of the months, and add the kids represented by balls. The result of the series of tests can be written in the title bar.



2. While playing ball, 10 kids in a first team throw their ball, all at the same time, at 10 kids in a second team and always hit a kid. (The balls do not affect each other.) The ones that are hit are eliminated. On the average, how many of the second team members remain untouched?

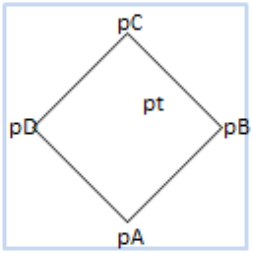
   Create a simulation with 100 random tests to determine this number experimentally. Now illustrate in a GPanel, for each attempt of the experiment, both teams as filled circles and draw the direction of the balls as lines. The result of the series of tests can be written in the title bar.



3. You can even determine the area of any given figure with the Monte Carlo simulation. Hold down the left mouse button to draw a freehanded outline. By clicking the right mouse button on a point anywhere inside of the outline, the area is filled and the simulation is carried out.

4. Conduct the chaos game with a square. Select the vertices pA(0, -10), pB(10, 0), pC(0, 10), pD(-10, 0) and any point *pt* on the inside.

   Divide the line segments between a randomly chosen vertex and *pt* with a division factor of 0.45 (*pt = getDividingPoint(corner, pt, 0.45)*).



3.11 IMAGE PROCESSING

INTRODUCTION

We understand a picture as a flat, rectangular area on which there are colored forms. In printing and computer technology, one describes an image as a grid-like arrangement of colored dots called **pixels**. The number of pixels per unit of area is called the **image resolution** and it is often indicated in *dots per inch* (dpi).

In order to save and process an image on the computer, the color must be defined as a number. There are several possibilities for this, which are called either **color metrics** or **color models**. One of the most popular models used is the RGB color model, where the intensity of the three color components red, green, and blue are represented by numbers between 0 (dark) and 255 (light) **more...**]. The ARGB model includes even another number between 0 and 255 that is the measure of transparency (alpha value) [**more...**].

In short: A computer image consists of a rectangular array of pixels that are encoded as colors. This is often called a **bitmap**.

PROGRAMMING CONCEPTS: *Image digitalization, resolution, color model, bitmap, image format*

COLOR MIXING IN THE RGB MODEL

TigerJython provides you with objects of the type *GBitmap*, to simplify your work with bitmaps. Using **bm = GBitmap(width, height)** you produce a bitmap with the desired number of horizontal and vertical pixels. Afterwards, you can set the color of the individual pixels using the method **setPixelColor(x, y, color)** and read them using *getPixelColor(x, y)*. With the method *image()* you can finally represent the bitmap in GPanel. Your program will draw the famous 3 circles of additive color mixing as you run through the bitmap with a nested for loop.

```
from gpanel import *

xRed = 200
yRed = 200
xGreen = 300
yGreen = 200
xBlue = 250
yBlue = 300

makeGPanel(Size(501, 501))
window(0, 501, 501, 0)    # y axis downwards
bm = GBitmap(500, 500)
for x in range(500):
  for y in range(500):
      red = green = blue = 0
      if (x - xRed) * (x - xRed) + (y - yRed) * (y - yRed) < 16000:
         red = 255
      if (x - xGreen) * (x - xGreen) + (y - yGreen) * (y - yGreen) < 16000:
         green = 255
```

```
      if (x - xBlue) * (x - xBlue) + (y - yBlue) * (y - yBlue) < 16000:
         blue = 255
      bm.setPixelColor(x, y, makeColor(red, green, blue))

image(bm, 0, 500)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Colors are defined by their red, green, and blue components. *makeColor(red, green, blue)* puts these color components together to a color (a color object).

For images we typically use an integer coordinate system with the origin in the upper left corner, with the positive y-axis pointing down [**more...**].

MAKING A GRAYSCALE IMAGE

At some point, you may have been asked how your image processing software (such as Photoshop, etc.) actually works. Here, you will get to know some of the simple procedures. Your program can turn a color image into a grayscale image by determining the average of the red, green, and blue components, and then use these to define the gray value.

```
from gpanel import *

size = 300

makeGPanel(Size(2 * size, size))
window(0, size, size, 0)    # y axis downwards
img = getImage("sprites/colorfrog.png")
w = img.getWidth()
h = img.getHeight()
image(img, 0, size)
for x in range(0, w):
    for y in range(0, h):
        col = img.getPixelColor(x, y)
        red = col.getRed()
        green = col.getGreen()
        blue = col.getBlue()
        intensity = (red + green + blue) // 3
        gray = makeColor(intensity, intensity, intensity)
        img.setPixelColor(x, y, gray)
image(img, size / 2, size)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can determine the color values as integers from a color object using the methods **getRed()**, *getGreen(), getBlue().*

The background has to be white, not transparent. If you want to allow transparency, you can determine the transparency value with *alpha = getAlpha()* and then use it in a extra parameter of *makeColor(red, green, blue, alpha)*.

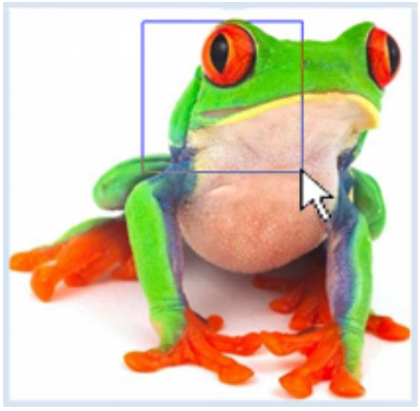
REUSABILITY

In many image processing programs the user has to be able to select a portion of the image. For this, you can create a temporary rectangle by clicking and dragging the mouse (a "rubber band rectangle"). Once you release the mouse button, the rectangular area will be definitely chosen. It is smart to solve this subproblem first, since its code can be used again later in many other image processing applications. Reusability is a sign of quality in all software development.

As you saw earlier, you can regard the drawing of the rubber band lines as an animation. In this case, however, the entire image needs to be rebuilt with each movement. A neat trick to avoid this is the **XOR drawing mode**. In this mode, a new figure is combined with the one underneath it so that the figure will be deleted again upon further repainting, without changing the underlying image. The disadvantage to this process is that the colors change while the figure is drawn. However, this does not usually matter in connection with rubber band rectangles.

The program framework should only call the function **doIt()** after the rectangle selection, and write the coordinates of the upper left corner *ulx (upper left x)*, *uly (upper left y)* and the lower right corner *lrx (lower right x), lry (lower right y)*. You will later insert your code for image processing in *doIt()*.

You should be able to understand the code with your previous experience in the chapter about mouse events without any major problems.



```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode(Color.white) # set XOR paint mode
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode() # establish normal paint mode
    ulx = min(x1, x2)
```

```
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)
    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    print "ulx = ", ulx, "uly = ", uly
    print "lrx = ", lrx, "lry = ", lry

x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can get the bitmap for a picture that you have already stored on your computer by using **getImage()**, where you must specify the fully qualified name, or just a part of the path relative to the directory in which your program is located. For images located in the distribution, you use the directory name *sprites*.

In the press event, you put the system into **XOR mode**, so that in your drag event handling you can first delete the old rectangle by drawing twice, and then draw the new one. You must store the vertices in the global variables *x1, y1, x2, y2*. If you draw the rubber band rectangle again with the release event before you switch to paint mode, the rectangle will disappear. If you switched to paint mode first, the rectangle would stay.

The program will work no matter how you decide to draw the rectangle. It always returns the correct values for *ulx,uly* and *lrx, lry* (always *ulx* < *lrx, uly* < *lry*). Be aware that you do not need to convert the mouse coordinates to window coordinates, since both are equal if you are using the same values for the window size with *size()* and the coordinate system with *window()*.

You still get drag events if you move the mouse out of the window. You have to be careful of what you do with such coordinates, otherwise the program could crash unexpectedly.

RED-EYE EFFECT

Image processing plays a central role in the post-processing of digital photos. There are numerous post-processing programs on the Internet, but you do not need to rely on them because you can now write your own program that will be better suited to your needs, with *Python* and a healthy degree of imagination and perseverance. Your task below is to write a program that can fix the red-eye effect. This occurs when the back of the eye (fundus) reflects the flash. Here you will use a picture of a frog, since it also has other red spots.

```
from gpanel import *
```

```
size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode("white")
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode()
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)

    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    for x in range(ulx, lrx):
        for y in range(uly, lry):
            col = img.getPixelColor(x, y)
            red = col.getRed()
            green = col.getGreen()
            blue = col. getBlue()
            col1 = makeColor(3 * red // 4, green, blue)
            img.setPixelColor(x, y, col1)
    image(img, 0, size)

x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The code for processing the image is latched in the function **doIt()** You can take everything else unchanged from the previous program. You can adjust the degree of attenuation of the color red. Here, the **red intensity** is set down to 75%. Be aware of the double slash, which performs an integer division (the remainder is ignored). The result is again an integer, just as it should be.

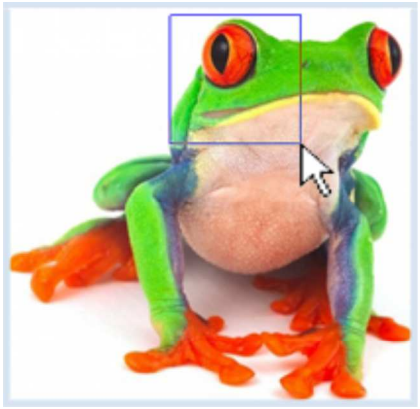
The program still shows some errors which you can easily fix. Firstly, it also discolors non-red areas, and secondly, it crashes when you pull the rubber band rectangle out of the window.

Of course it would be really nice if the program could find the red eyes itself. However, to do

this it would have to analyze the image and recognize its contents automatically, which is an especially challenging problem in computer science [**more...**].

CUTTING AND STORING PICTURES

Cutting images is also one of the basic functions of image processing programs. Your program can not only copy a selected part of the image to another window using the rubber band rectangle, but it can also store this image as a JPEG file for future use.



```
from gpanel import *

size = 300

def onMousePressed(e):
    global x1, y1
    global x2, y2
    setColor("blue")
    setXORMode("white")
    x1 = x2 = e.getX()
    y1 = y2 = e.getY()

def onMouseDragged(e):
    global x2, y2
    rectangle(x1, y1, x2, y2) # erase old
    x2 = e.getX()
    y2 = e.getY()
    rectangle(x1, y1, x2, y2) # draw new

def onMouseReleased(e):
    rectangle(x1, y1, x2, y2) # erase old
    setPaintMode()
    ulx = min(x1, x2)
    lrx = max(x1, x2)
    uly = min(y1, y2)
    lry = max(y1, y2)
    doIt(ulx, uly, lrx, lry)

def doIt(ulx, uly, lrx, lry):
    width = lrx - ulx
    height = lry - uly
    if ulx < 0 or uly < 0 or lrx > size or lry > size:
        return
    if width < 20 or height < 20:
        return

    cropped = GBitmap.crop(img, ulx, uly, lrx, lry)
    p = GPanel(Size(width, height))  # another GPanel
    p.window(0, width, 0, height)
    p.image(cropped, 0, 0)
    rc = save(cropped, "mypict.jpg", "jpg")
    if rc:
        p.title("Saving OK")
    else:
        p.title("Saving Failed")
```

```
x1 = y1 = 0
x2 = y2 = 0

makeGPanel(Size(size, size),
    mousePressed = onMousePressed,
    mouseDragged = onMouseDragged,
    mouseReleased = onMouseReleased)
window(0, size, size, 0)    # y axis downwards

img = getImage("sprites/colorfrog.png")
image(img, 0, size)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can view more than one GPanel window if necessary, by creating GPanel objects. To draw, use the graphics commands which you call using the point operator.
If the selected section is too small (especially if you click with the mouse without dragging), *doIt()* ends with an empty *return*, and likewise if the vertices are not in the image area.

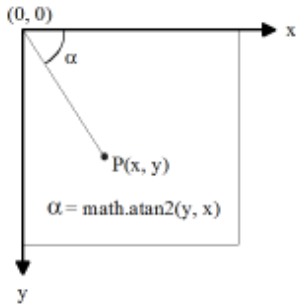
To save, use the method **save()**, where the last parameter determines the image format. The allowed values are: "bmp", "gif", "jpg", "png".

EXERCISES

1. Write a program that swaps the red and green components of the image *colorfrog.png*.

2. Write a program where you can rotate the image by dragging the mouse. Use the function *atan2(y, x)* which provides you with the angle α to the point *P(x, y)*. You still have to convert this to degrees using *math.degrees()* before you can rotate the picture with *GBitmap.scale()*.

   You can take *colorfrog.png* as a test image again.



3. Write a photo retouching program that can store the color of a pixel with a click of the mouse (color picker). The following dragging should draw colored circles filled this way into the image. Here you have to use the press, drag, and click events. You can again use *colorfrog.png* as a test image. Write the 3 color components of the "picked" color in the title bar of the window.

EXTRA MATERIAL: FILTERING IMAGES WITH CONVOLUTION

You surely know that in conventional image processing programs you are able to modify an image with various filters, such as smoothing filters, sharpening filters, blurring filters, etc. Here, the important principle of **convolution** is used, which you can learn about [**more...**]. In this process, you change the color values of each pixel by calculating a new value from it and its eight neighboring pixels, according to a filtering rule.

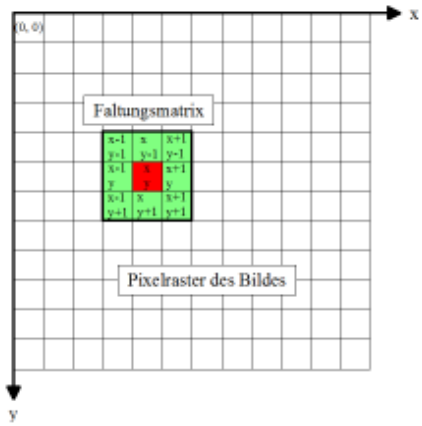
In detail, this works as follows: For the sake of simplicity, consider a greyscale image where each pixel in the RBG coloring model possesses a gray value *v* between 0 and 255. The filtering rule is defined by nine numbers that are arranged in a square:

m00 m01 m02
m10 m11 m12
m20 m21 m22

This representation is called a **convolution matrix** (also called **mask**). In *Python* we implement it line by line in a list

mask = [[0, -1, 0], [-1, 5, 1], [0, -1, 0]]

With this data structure you can easily access the individual values with double indices, for example m12 = mask[1][2] = 1. These nine numbers are weighting factors for a pixel and its eight neighbors. Now you can calculate the new gray value *vnew* of a pixel at the point *x, y* from the existing nine values v(x, y) as follows:



vnew(x, y) = m00 * v(x - 1, y -1) + m01 * v(x, y - 1) + m02 * v(x + 1, y - 1) +
m10 * v(x - 1, y) + m11 * v(x, y) + m12 * v(x + 1, y) +
m20 * v(x - 1, y + 1) + m21 * v(x , y + 1) + m22 * v(x + 1, y + 1)

To illustrate, one could say that for the recalculation one places the convolution matrix above the pixel, multiplies its values with the underlying gray values, and finally sums them all up. The program performs these convolution operations for all of the pixels (except the boundary points) and then saves the resulting gray values in a new bitmap, which it then displays. To do this you move the convolution matrix row by row, from left to right and from top to bottom, over the image with a *for* structure. Here you use the convolution matrix values of a sharpening filter and the grayscale image *frogbw.png* of the frog.

```
from gpanel import *

size = 300

makeGPanel(Size(2 * size, size))
window(0, size, size, 0)    # y axis downwards

bmIn = getImage("sprites/frogbw.png")
image(bmIn, 0, size)
w = bmIn.getWidth()
h = bmIn.getHeight()
bmOut = GBitmap(w, h)

#mask = [[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]]  # smoothing
mask = [[ 0, -1,  0], [-1,  5, -1], [0,  -1,  0]] #sharpening
#mask = [[-1, -2, -1], [ 0,  0,  0], [ 1,  2,  1]] #horizontal edge extraction
#mask = [[-1,  0,  1], [-2,  0,  2], [-1,  0,  1]] #vertical edge extraction
```

```
for x in range(0, w):
    for y in range(0, h):
        if x > 0 and x < w - 1 and y > 0 and y < h - 1:
            vnew = 0
            for k in range(3):
                for i in range(3):
                    c = bmIn.getPixelColor(x - 1 + i, y - 1 + k)
                    v = c.getRed()
                    vnew +=  v * mask[k][i]
            # Make int in 0..255
            vnew = int(vnew)
            vnew = max(vnew, 0)
            vnew = min(vnew, 255)
            gray = Color(vnew, vnew, vnew)
        else:
            c = bmIn.getPixelColor(x, y)
            v = c.getRed()
            gray = Color(v, v, v)

        bmOut.setPixelColor(x, y, gray)

image(bmOut, size / 2, size)
```

MEMO

In a convolution, each pixel is replaced by a weighted average of itself and its neighboring points. The filter type is determined by the convolution matrix. You can experiment with the following well-known convolution matrices, or you can invent your own.

Filter type	Convolution matrix
Sharpening filter	$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$
Smoothing filter	$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix}$
Edge extraction (horizontal)	$\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$
Edge extraction (vertical)	$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$

3.12 PRINTING IMAGES

INTRODUCTION

You have already learned how to let the turtle draw on a high resolution printer in the chapter Turtle Graphics. You can similarly render an image from GPanel on the printer. You can also use a virtual printer that creates a graphic file in high resolution format (such as Tiff or EPS) [**more...**]. To do this, you define a parameterless function with any name such as *doIt()* that will contain all of the commands necessary to create the image. With a direct call, the image will appear on the screen. To print it, call *printerPlot(doIt)*. You can also specify a scaling factor *k*, and if you do, call *printerPlot(doIt, k)* instead. If k < 1 it results in a reduction, and if k >1 it results in an enlargement.

PROGRAMMING CONCEPTS: *High resolution graphic*

ROSETTES

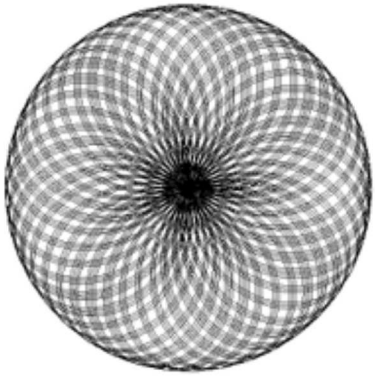
The rose-like curves go all the way back to the 18th century to the mathematician Guido Grandi [**more...**]
The generating functions are most easily expressed using polar coordinates (ρ, φ). It has a parameter *n*:

$$\rho = \sin(n\varphi)$$

The Cartesian coordinates are obtained as usual:

$$x = \rho \cos(\varphi)$$
$$y = \rho \sin(\varphi)$$

You get a pretty rosette using $n = \sqrt{2}$. However, it looks even nicer on a printer than it does on the screen.



```
from gpanel import *
import math

def rho(phi):
    return math.sin(n * phi)

def doIt():
    phi = 0
    while phi < nbTurns * math.pi:
        r = rho(phi)
        x = r * math.cos(phi)
        y = r * math.sin(phi)
        if phi == 0:
          move(x, y)
        else:
          draw(x, y)
        phi += dphi

n = math.sqrt(2)
dphi = 0.01
nbTurns = 100
makeGPanel(-1.2, 1.2, -1.2, 1.2)
```

```
doIt()
printerPlot(doIt)
```

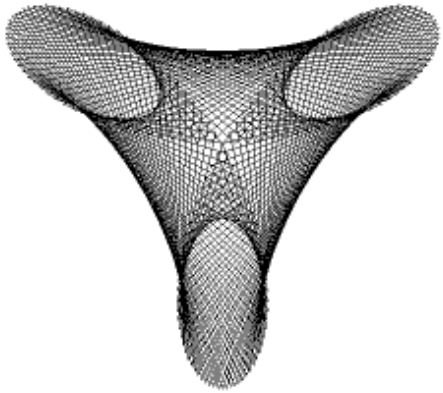
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Depending on the choice of the parameter *n*, you can create different kinds of curves. Try it with natural numbers, rational numbers (fractions), and with irrational numbers (π, e).

MAURER ROSES

The mathematician Peter Maurer introduced these curves in 1987 in his article "A Rose is a Rose...". They use rosettes as "guidelines". From this guideline you repeatedly choose points, after a specific rotation angle *d*, 360 points in total. Afterwards, you connect these points with straight lines.



Depending on the choice of *n* and *d*, completely different curve shapes can be created. Print them to make them look even nicer (in this example, n = 3 and d = 47 degrees).

```
from gpanel import *
import math

def sin(x):
    return math.sin(math.radians(x))

def cos(x):
    return math.cos(math.radians(x))

def cartesian(polar):
    return [polar[0] * cos(polar[1]), polar[0] * sin(polar[1])]

def rho(phi):
    return sin(n * phi)

def doIt():
    for i in range(361):
        k = i * d
        pt = [rho(k), k]
        corners.append(pt)

    move(cartesian(corners[0]))
    for pt in corners:
        draw(cartesian(pt))

corners = []
n = 3
d = 47
makeGPanel(-1.2, 1.2, -1.2, 1.2)
doIt()
printerPlot(doIt)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In the program you use degrees, not radians. Therefore, it is convenient to define your own functions for sine and cosine that calculate with degrees. This also simplifies the notation, since you will not always need to write *math.* beforehand.

Likewise, it is convenient to make a conversion from polar to Cartesian coordinates in the function *cartesian()*, where the coordinate pairs are packaged as a list.

Save the polar coordinates of the 361 points which you select from the guideline in the list *corners*. In the end, you run through them and draw lines from point to point using *draw()*.
You can draw other known Maurer roses with the following parameters:

n	d
2	39
2	31
6	71

EXERCISES

1. Draw 50 concentric circles with the function *wave(center, wavelength)* with *center* as the midpoint and *wavelength* as the radius increment. One could interpret the image as the peaks of a circular wave. Draw the wave with a slightly displaced center and then look at the resulting interference image on a printout. What curve known from geometry can you recognize?

3.13 WIDGETS

INTRODUCTION

Programs you know usually have a graphical user interface (GUI). You will recognize a menu bar, input fields, and buttons. GUI components, also called **widgets**, are regarded as objects, which you already know from the chapter *Turtle Objects*. If you want to write a program with a modern user interface, it is essential that you know and understand the basic concepts of object-oriented programming (OOP) [**more...**].

The widgets are divided into different classes according to the following list:

Widget	**Klasse**
Buttons	JButton
Labels	JLabel
Text fields	JTextField
Menu bars	JMenuBar
Menu items	JMenuItem
Menus with menu items	JMenu

Just like you generated a turtle by calling the constructor of the class *turtle*, you need to create a GUI component by calling the corresponding class constructor. The constructors often have parameters with which you can set certain properties of the widget. For example, you can create an input field with a length of 10 characters using *tf = JTextField(10)*.

When calling the constructor, you also need to define a variable that you will use later to access the object. For example, *tf.getText()* returns the text currently in the text field *tf*.

In order to make a widget visible in GPanel, you use the function *addComponent()* and provide it with the object variable. The widgets are automatically placed in the order of the calls in the upper part of the GPanel window [**more...**].

PROGRAMMING CONCEPTS: *Graphical user interface, GUI component, GUI callback*

PI WITH THE RAINDROP SIMULATION

You have already learned how to determine an area using the Monte Carlo simulation. Imagine that you are drawing a quarter circle with a radius of 1 into a square with a side length of 1. If you now let *n* raindrops fall evenly onto the square you will easily be able to figure out how many of them fall onto of the quarter circle on average.

Since the area of the quarter circle is

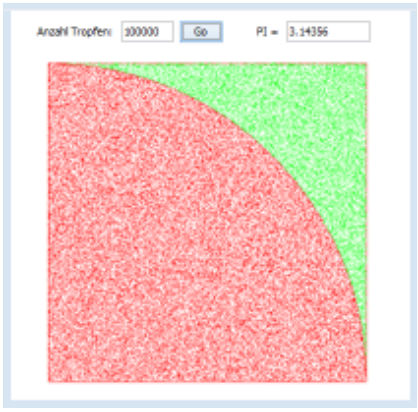
$$S = \frac{1}{4} * r^2 * \pi = \frac{\pi}{4}$$

and the area of the square is 1, the number of drops should be

$$k = n * \frac{\pi}{4}$$

So, if in a computer simulation you let *n* drops fall and count k, you get an approximation of Pi with

$$\pi = \frac{4 * k}{n}$$



The GUI consists of two labels, two text boxes, and a button. Once created, you add them to GPanel with **addComponent()**.

It should be clear that clicking on the OK button can be considered an event. The callback is registered via the parameter named **actionListener** in the constructor of *JButton*. Hopefully you remember that you should not execute lengthy code in a callback. So, you merely call **wakeUp()** in the callback, whereby the program, which was halted in the *while* loop using **putSleep()**, is awakened and then executes the simulation.

```
from gpanel import *
import random
from javax.swing import *

def actionCallback(e):
    wakeUp()

def createGUI():
    addComponent(lbl1)
    addComponent(tf1)
    addComponent(btn1)
    addComponent(lbl2)
    addComponent(tf2)
    validate()

def init():
    tf2.setText("")
    clear()
    move(0.5, 0.5)
    rectangle(1, 1)
    move(0, 0)
    arc(1, 0, 90)

def doIt(n):
    hits = 0
    for i in range(n):
        zx = random.random()
        zy = random.random()
        if zx * zx + zy * zy < 1:
            hits = hits + 1
            setColor("red")
        else:
            setColor("green")
        point(zx, zy)
    return hits

lbl1 = JLabel("Number of drops: ")
lbl2 = JLabel("             PI = ")
tf1 = JTextField(6)
tf2 = JTextField(10)
btn1 = JButton("OK", actionListener = actionCallback)

makeGPanel("Monte Carlo Simulation", -0.1, 1.1, -0.1, 1.1)
createGUI()
```

```
tf1.setText("10000")
init()

while True:
    putSleep()
    init()
    n = int(tf1.getText())
    k = doIt(n)
    pi =  4 * k / n
    tf2.setText(str(pi))
```

MEMO

Widgets are objects of the Swing class library. They are created with the constructor that has the name of the class. When calling the constructor, you define a variable with which you can access the object. To display the widget in the GPanel, call the function **addComponent()** and pass it the widget variable.

After you have added all of the widgets to the GPanel, you should call **validate()** so that the window is rebuilt with the inserted widgets with certainty. You can register button callbacks with the parameter named **actionListener**. Remember that a callback should never execute lengthy code.

MENUS (but not for food!)

Many windows have a menu bar with many menu items. When you click on a menu item, a submenu can also be opened, which in turn contains menu items. Menus and menu items are also regarded as objects. The selection of a menu option triggers an event that is handled by a callback.

You build a menu by creating a **JMenuBar()** object and adding *JMenuItem* objects to it using *add()*. You can also add a submenu. For this, create a *JMenu* object, and add *JMenuItem* objects to it. Thus, a menu is built hierarchically.

In order to simplify the code a bit, you can use the same callback *actionCallback()* for all menu options. Register it with each constructor of **JMenuItem** with the parameter **actionPerformed**. In the callback, you can determine by which menu option the callback was triggered with **getSource()**.


```
from gpanel import *
from javax.swing import *

def actionCallback(e):
    if e.getSource() == goItem:
        wakeUp()
    if e.getSource() == exitItem:
        dispose()
    if e.getSource() == aboutItem:
        msgDlg("Pyramides Version 1.0")

def doIt():
    clear()
    for i in range(1, 30):
        setColor(getRandomX11Color())
        fillRectangle(i/2, i - 0.35, 30 - i/2, i + 0.35)
```

```
fileMenu = JMenu("File")
goItem = JMenuItem("Go", actionPerformed = actionCallback)
exitItem = JMenuItem("Exit", actionPerformed = actionCallback)
fileMenu.add(goItem)
fileMenu.add(exitItem)

aboutItem = JMenuItem("About", actionPerformed = actionCallback)

menuBar = JMenuBar()
menuBar.add(fileMenu)
menuBar.add(aboutItem)

makeGPanel(menuBar, 0, 30, 0, 30)

while not isDisposed():
    putSleep()
    if not isDisposed():
        doIt()
```

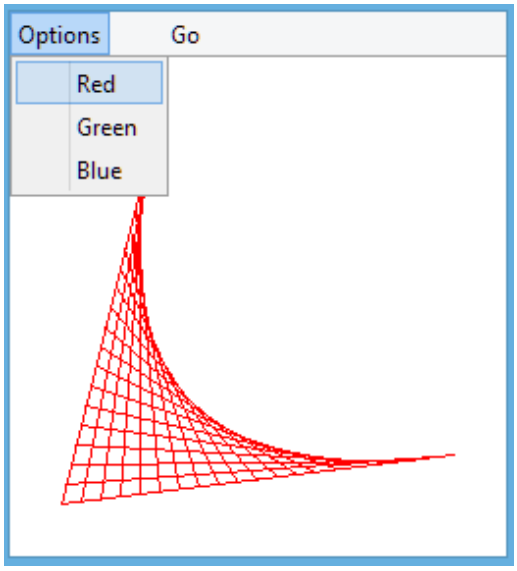
MEMO

Remember to follow the rule that a callback should never execute lengthy code. You therefore perform the drawing in the main block. To ensure that your program terminates with certainty after you press the close button of the window or the exit option, use *isDisposed()* to test whether the window was closed [**more...**].

EXERCISES

1. Edit the program Moiré from chapter 3.2 and add a text label, an input field for the delay time, and an OK button. When you click on the OK button, the image will be recreated with the specified delay time (in milliseconds).

2. Edit the program under "Elegant Thread Graphic Algorithms" in chapter 3.8 and add the following menu: The menu item "Options" should contain a submenu with the text "Red", "Green", and "Blue". The menu item "Go" should draw the thread graphics with the color selected under Options. If no color is chosen yet, it will be drawn using black.



3*. Take one of your favorite programs from the GPanel graphic and add some useful widgets to it.

Documentation GPanel

Module import: from gpanel import *

Function	Action
makeGPanel()	creates a GPanel graphics window with coordinates (x = 0..1, y = 0..1). Current cursor at (0, 0)
makeGPanel(xmin, xmax, ymin, ymax)	creates a GPanel graphics window with given float coordinate system. Current cursor at (0, 0)
makeGPanel(xmin, xmax, ymin, ymax, False)	same as above, but invisible window (call visible(True), to make it visible)
makeGPanel(Size(width, height))	same as makeGPanel(), but window size user selectable (in pixels)
getScreenWidth()	returns the screen width (in pixels)
getScreenHeight()	returns the screen height (in pixels)
window(xmin, xmax, ymin, ymax)	sets a new coordinate span
drawGrid(x, y)	draws a coordinate grid with 10 ticks in range 0..x, 0..y. Label text depends if x, y or int or float
drawGrid(x, y, color)	same with given grid color
drawGrid(x1, x2, y1, y2)	same with given span x1..x2, y1..y2
drawGrid(x1, x2, y1, y2, color)	same with given grid color
drawGrid(x1, x2, y1, y2, x3, y3)	same with given number of ticks x3, y3 in x- and y-direction
drawGrid(x1, x2, y1, y2, x3, y3, color)	same with given grid color
drawGrid(p, ...)	same as drawGrid() with given GPanel references (for several GPanels)
visible(isVisible)	shows/hides the window
resizeable(isResizeable)	makes the window resizable (default: not resizeable)
dispose()	closes the window and releases resources
isDisposed()	returns True, if window is disposed by title bar's close button or by calling displose()
bgColor(color)	sets background color (X11 color string or Color type returned my makeColor())
title(text)	sets text in title bar
makeColor(colorStr)	returns color as Color type that corresponds to given X11 color string
windowPosition(ulx, uly)	sets screen position (in pixels)
windowCenter()	sets the window in the center of the screen
storeGraphics()	stores the current graphics in internal image buffer
recallGraphics()	renders the content of the internal image buffer
clearStore(color)	erases the internal image buffer by painting it with given color
delay(time)	pauses the program execution for given amount of time (in ms)
getDividingPoint(pt1, pt2, ratio)	returns the point that divides the line from pt1 to pt2 with the given ratio (may be negative and greater than 1)
getDividingPoint(c1, c2, ratio)	same with complex
clear()	clears the graphics window and sets the graphics cursor to (0, 0)
erase()	clears the graphics window without changing the position of the graphics cursor
putSleep()	pauses program execution until wakeUp() is called
wakeUp()	resumes paused program execution

linfit(X, Y)	performs a linear regression y = a*x + b with data in X- and Y-lists and returns (a, b)

Drawing

lineWidth(width)	sets the line width (in pixel)
setColor(color)	sets die drawing color (X11 color string or Color type)
move(x, y)	moves cursor to (x, y) without drawing a line
move(coord_list)	moves cursor to point list [x, y] without drawing a line
move(c)	moves cursor to complex(x, y) without drawiing a line
getPosX()	returns the cursor's current x-coordinate
getPosY()	returns the cursor's current y-coordinate
getPos()	returns the cursor current x-, y-coordinates as list
draw(x, y)	draws line to (x, y) and updates cursor
draw(coord_list)	draws line to [x, y] and updates cursor
draw(c)	draws line complex [x, y] and updates cursor
line(x1, y1, x2, y2)	draws line from (x1, y1) to (x2, y2) without modifying cursor
line(pt1, pt2)	draws line from pt1 = [x1, y1] to pt2 = [x2, y2] without modifying cursor
line(c1, c2)	draws line complex(x1, y1) to complex(x2, y2) without modifying cursor
circle(radius)	draws circle with center at current cursor position and given radius
fillCircle(radius)	draws fiilled circle with center at current cursor position and given radius (fill color = pen color)
ellipse(a, b)	draws ellipse with center at current cursor positon and given semiaxis
fillEllipse(a, b)	draws ellipse with center at current cursor positon and given semiaxis (fill color = pen color)
rectangle(a, b)	draws rectangle with center at current cursor position and given width and height
rectangle(x1, y1, x2, y2)	draws rectangle with center at current cursor position and given diagonal
rectangle(pt1, pt2)	same with diagonal point lists
rectangle(c1, c2)	same with diagonal complex
fillRectangle(a, b)	draws filled rectangle with center at cursor and given width and height (fill color = pen color)
fillRrectangle(x1, y1, x2, y2)	draws filled rectangle with center at cursor and given diagonal (fill color = pen color)
fillRectangle(pt1, pt2)	same with diagonal point lists
fillRrectangle(c1, c2)	same with diagonal complex
arc(radius, startAngle, extendAngle)	draws arc with center at cursor and given radius, start and sector angle (0 to east, positive counterclockwise)
fillArc(radius, startAngle, extendAngle)	same, but filled (fill color = pen color)
polygon(x_list, y_list)	draws polygon with vertexes from x_list and y_list
polygon((li[pt1, pt2,..)	same with list of point lists pt1, pt2,...
polygon(li[c1, c2, c3,...])	same with list of complex c1, c2,...
fillPolygon(x_list, y_list)	draws filled polygon with vertexes from x-list and y-list (fill color = pen color)
fillPolygon((li[pt1, pt2,..)	same with list of point lists pt1, pt2,...
fillPolygon(li[c1, c2, c3,...])	same with list of complex c1, c2,...
quadraticBezier(x1, y1, xc, yc, x1, y2)	draws quadratic Bezier-curve with 2 points (x1, y1), (x2, y2) and control point (xc, yc)

quadraticBezier(pt1, pc, pt2)	same with point lists
quadraticBezier(c1, cc, c2)	same with complex
cubicBezier(x1, y1, xc1, yc1, xc2, yc2, x2, y2)	draws cubic Bezier-curve with 2 points (x1, y1), (x2, y2) and two control points (xc1, yc1), (yc2, yc2)
cubicBezier(pt1, ptc1, ptc2, pt2)	same with point lists
cubicBezier(c1, cc1, cc2, c2)	same with complex
triangle(x1, y1, x2, y2, x3, y3)	draws a triangle with vertexes from x-, y-coordinates
triangle(pt1, pt2, pt3)	same with point lists
triangle(li[pt1, pt2, pt3])	same with list of point lists
triangle(c1, c2, c3)	same with complex
fillTriangle(x1, y1, x2, y2, x3, y3)	draws a filled triangle with vertexes from x-, y-coordinates (fill color = pen color)
fillTriangle(pt1, pt2, pt3)	same with point lists
fillTriangle(li[pt1, pt2, pt3])	same with list of point lists
fillTriangle(c1, c2, c3)	same with complex
point(x, y)	draws one single point (pixel) at (x, y)
point(pt)	same with point list
point(complex)	same with complex
fill(x, y, color, replacementColor)	fills a closed area with point (x, y) inside by replacing each pixel with given color by a pixel with replacementColor (floodfill)
fill(pt, color, replacementColor)	same with point list
fill(complex, color,replacementColor)	same with complex
image(path, x, y)	shows image in GIF , PNG oder JPG format at lower-left position x, y. The image path may be relative to the TigerJython folder, in the distribution JAR (folder *sprites*) or a URL starting with http://
image(path, pt)	same with point list
image(path, complex)	same with complex
imageHeighpath)	returns the height of the image
imageWidth(path)	returns the width of the image
enableRepaint(boolean)	enables/disables automatic rendering of the offscreen buffer (default: enabled)
repaint()	renders the offscreen buffer on screen (necessary if the automatic rendering is disabled)
setPaintMode()	selects normal painting by overwriting the background
setXORMode(color)	selects XOR-painting with given color. Painting twice removes without artefact.
getPixelColor(x, y)	returns color of pixel at (x, y) as Color type
getPixelColor(pt)	same with point list
getPixelColor(complex)	same with complex
getPixelColorStr(x, y)	returns color of pixel at (x, y) as X11 color string
getPixelColorStr(pt)	same with point list
getPixelColorStr(complex)	same with complex

Text

text(string)	displays text starting at current cursor position
text(x, y, string)	display text starting at given x-, y-coordinates
text(pt, string)	same with point list
text(complex, string)	same with complex

text(x, y, string, font, textColor, bgColor)	displays text at given x-, y-coordinates with given font, text and background color
text(pt, string, font, textColor, bgColor)	same with point list
text(complex,string, font, textColor, bgColor)	same with complex
font(font)	selects another text font (see below for font format)

Callbacks

makeGPanel(mouseNNN = onMouseNNN) auch mehrere, durch Komma getrennt	registers the callback function onMouseNNN(x, y) that is called when a mouse event happens. Values for NNN: Pressed, Released, Clicked, Dragged, Moved, Entered, Exited, SingleClicked, DoubleClicked
isLeftMouseButton(), isRightMouseButton()	returns True, if the event is caused by the left/right mouse button
makeGPanel(keyPressed = onKeyPressed)	registers the callback onKeyPressed(keyCode) that is called when a keyboard key is hit. keyCode is a unique integer value that identifies the key
getKeyModifiers()	returns an integer code for special keyboard keys (shift, ctrl, etc., also combined)
makeGPanel(closeClicked = onCloseClicked)	registers the callback onCloseClicked() that is called when the title bar close button is hit. The window must be closed manually by calling dispose()

Keyboard

getKey()	returns the character (as string) of the last key pressed
getKeyCode()	returns the key code of the last key pressed
getKeyWait()	stops the program until a key is pressed and returns the charactor (as string) of the key
getKeyCodeWait()	stops the program until a key is pressed and returns the key code of the key
kbhit()	returns True, if a key was hit since the last call of getKey() or getKeyCode()

GUI Components

add(component)	inserts a GUI component near the top border of the window
validate()	repaints the window after a component has been added

Status Bar

addStatusBar(height)	adds a status bar at the bottom of the window with given height (in pixels)
setStatusText(text)	displays text in the status bar (old text is erased)
setStatusText(text, font, color)	displays text in the status bar with given font and color (old text is erased)

Font Format

Font(name, style, size)	creates a new font with given name, style and size
name	a string with a font name available on the system, e.g. "Times New Roman", "Arial", "Courier"
style	One of the stype constants: Font.PLAIN, Font.BOLD, Font.ITALIC, may also be combined: Font.BOLD + Font.ITALIC
size	an integer with an available font size in pixels, e.g. 12, 16, 72

Dialogs

msgDlg(message)	opens a modal dialolg with an OK button and given message
msgDlg(message, title = title_text)	same with title text

inputInt(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns integer (the dialog is shown again, if no integer is entered). Cancel or Close terminate the program
inputInt(prompt, False)	same, but Cancel/Close do not terminate, but returns None
inputFloat(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns float (the dialog is shown again, if no float is entered). Cancel or Close terminate the program
inputFloat(prompt, False)	same, but Cancel/Close do not terminate, but returns None
inputString(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns string. Cancel or Close terminate the program
inputString(prompt, False)	same, but Cancel/Close do not terminate, but returns None
input(prompt)	opens a modal dialog with OK/Cancel buttons. OK returns integer, float or string. Cancel or Close terminate the program
input(prompt, False)	same, but Cancel/Close do not terminate, but returns None
askYesNo(prompt)	opens a modal dialog with Yes/No buttons. Yes returns True, No returns False. Cancel or Close terminate the program
askYesNo(prompt, False)	same, but Close do not terminate, but returns None

Module import: from fitter import *
Curve fitting:

polynomfit(xdata, ydata, n)	fits a polynom of order n and returns the fitted values in ydata. Return value: list with n + 1 polynom coefficients
splinefit(xdata, ydata, nbKnots)	fits a spline function that passes through nbKnots aequidistant data points. Returns the fitted data in ydata
functionfit(func, derivatives, initialGuess, xdata, ydata)	fits the function func(x, param) with n parameters in list param. derivatives(x, param) returns a list with the values of the partial derivatives to the n parameters. initGuess is a list with n guessed values for the n parameters
functionfit(func, derivatives, initialGuess, xdata, ydata, weights)	same but with a list weights that determines the relative weights of the data points
toAequidistant(xrawdata, yrawdata, deltax)	returns two lists xdata, ydata with aequidistant values separated by deltax (linear interpolation)

chapter four

SOUND

Learning Objectives

- You know how sound is digitized and stored.
- You are familiar with the term sampling and its implications.
- You can record a sound with your own program, specifically change it, play it back, and save it.

"As the skills that constitute literacy evolve to accommodate digital media, computer science education finds itself in a sorry state. While students are more in need of computational skills than ever, computer science suffers dramatically low retention rates and a declining percentage of women and minorities. Studies of the problem point to the over-emphasis in computer science classes on abstraction over application, technical details instead of usability, and the stereotypical view of programmers as loners lacking creativity. Media Computation, teaches programming and computation in the context of media creation and manipulation."

In Forte, Guzdial, Not Calculation: Media as a Motivation and Context for Learning

4.1 PLAYING BACK SOUND

INTRODUCTION

In order to process a sound signal in the computer, it must first be digitized. To do this, one samples it at equidistant time steps and converts the value of the signal to a number at every sampling time using an analog-to-digital converter. The sound signal then yields a sequence of numbers that can be stored and processed in the computer. The sampling frequency (or sampling rate) is understood as the number of samples per second. This is standardized for the WAV audio format and can have the following values: 8000, 11025, 16000, 22050 and 44100 Hertz. The higher the sampling frequency, the more precisely the sound can be restored by a digital-to-analog conversion. The value range of the samples is also important for the quality. In the *TigerJython* sound library, values are always stored as integers in a list in the 16-bit range (-32768 and 32767).

We also need to distinguish whether we are dealing with a monaural or a binaural sound. Depending on this, one or two channels are used. In the case of two channels (stereo), the values for the left and the right channel are stored as consecutive numbers.

In this chapter you will need a computer with a sound card, the ability to listen to sound through a speaker or headphones, and a microphone.

PROGRAMMING CONCEPTS: *Sound digitization, audio signal, sample, sampling rate*

LISTENING TO SOUND

Find a fun sound clip in WAV format that has a short duration (around 2 to 5 seconds long). You can look on the Internet. Copy the sound file under the name *mysound.wav* in the same directory as your program.

First, import all of the functions of the **sound library**. Next, copy the sound samples into the list **samples**and write the information of the sound file into the **console window**, for you need to know the sampling rate. In the example shown here, its value is **22050 Hz**. With **openMonoPlayer** you have the ability to play back the sound. If you enter the wrong sampling rate, the sound will be played with a different speed, hence with other frequencies.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
print getWavInfo("mysound.wav")

openMonoPlayer(samples, 22050)
play()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The function **getWavMono()** provides the sound samples in a *Python* list. Each value is an integer in the range between -32768 and 32767. The function **openMonoPlayer(**) provides a sound player so that the sound can be played with *play()*.

Since lists can only have a certain maximum size that depends on the memory capacity of your

computer, only relatively short sound clips can be read with *getWavMono()*.

SOUND WAVE

It is interesting to also represent sound samples graphically. To do this, simply use a **GPanel** window and run through the list in a **for loop**.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
print getWavInfo("mysound.wav")

openMonoPlayer(samples, 44100)
play()

from gpanel import *

makeGPanel(0, len(samples), -33000, 33000)
for i in range(len(samples)):
    draw(i, samples[i])
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

We have to choose the coordinate system of the GPanel conveniently. The values displayed in the x-direction are between 0 and the number of sampling values, which is equal to the length of the sample list. The values of the y-direction are between -32768 and 32767. This is why we use a range of +-33000.

THERE IS AN EASIER WAY

If you just want to play a sound file, you only need three lines of code. You can even play long sounds, for example your favorite songs.

```
from soundsystem import *

openSoundPlayer("myfavoritesong.wav")
play()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can also use some sound clips that come in the distribution of *TigerJython.* Choose any of the following file names:

Sound File	Description
wav/bird.wav	chirping bird
wav/boing.wav	boing
wav/cat.wav	meowing cat
wav/click.wav	click
wav/dummy.wav	empty sound
wav/explode.wav	explosion
wav/frog.wav	croaking frog
wav/mmm.wav	eating sound
wav/notify.wav	notification sound
wav/ping.wav	ping sound

(The list is constantly updated. If you have a WAV sound file with the same name in your own subdirectory *wav*, that one will be used.)

The sound player knows many control commands, just as a professional music player does. You can, for example, stop the song with *pause()* and then continue playing the song at the same spot using *play()*.

The duration of the sound is not limited in these functions because the sound is only read and played back in small packets (streaming player).

play()	plays back a sound from the current position and returns immediately
blockingPlay()	plays back a sound from the current position and waits until it is finished playing
advanceFrames(n)	moves forward from the current position by the given number of samples
advanceTime(t)	moves forward from the current position by the specified time
getCurrentPos()	returns the current position
getCurrentTime()	returns the current playing time
pause()	pauses playback. *play()* will start it again
rewindFrames(n)	moves backwards from the current position by the given number of samples
rewindTime(t)	moves backwards from the current position by the specified time
stop()	stops playback. The playback position is set to the beginning
setVolume(v)	adjusts the volume (value between 0 and 1000)

PLAYING MP3 SOUND FILES

To play sounds in the MP3 format, you will need additional library files which you can download and unzip separately **here**. Create the subdirectory *Lib* (if it does not already exist) in the directory where tigerjython2.jar is located, and then copy the unzipped files into it.

Instead of using *openSoundPlayer()*, *openMonoPlayer()*, and *openStereoPlayer()* for MP3 files use **openSoundPlayerMP3()**, *openMonoPlayerMP3()* and *openStereoPlayerMP3()* and indicate the path to the sound file. To play, use the same functions mentioned above.

```
from soundsystem import *

openSoundPlayerMP3("song.mp3")
play()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

To play MP3 sound files, you will need additional JAR library files that need to be located in the directory *Lib* of the home directory to tigerjython2.jar.

EXERCISES

1. Explain why the tone frequencies are changed if you change the sampling rate during playback

2. Show a sound wave in the GPanel representing a short range of 0.1 seconds starting at the 1 second mark. Explain the image.

3. Create a sound player with a GPanel where the following commands can be executed using the keyboard:

Key	Action
Cursor up	play
Cursor down	pause
Cursor left	rewind by 10 s
Cursor right	advance by 10 s
Buchstabe s	stop

   Write the command list as text in the window. With each key press, the action should be written out in the title line.

4.2 SOUND EDITING

INTRODUCTION

As you know, sound samples (sampling values of a sound) are stored in a list and can be played back again with this list. If you want to edit the sound, you can easily change the list accordingly.

PROGRAMMING CONCEPTS: *Rectangular wave, integer division, modulo operation*

CHANGING THE LOUDNESS/VOLUME

The program should reduce the volume of the sound by one quarter. To do this, copy the sound list to **another list**, where each list element is set to **¼** of its original value.

```
from soundsystem import *

samples = getWavMono("mysound.wav")
soundlist = []
for item in samples:
    soundlist.append(item // 4)

openMonoPlayer(soundlist, 22010)
play()
```

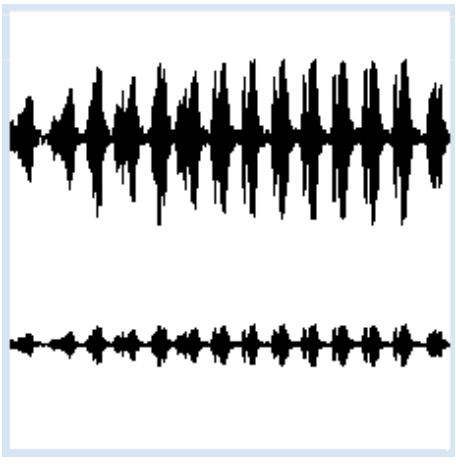
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

To copy a list, first create an **empty list** then fill it using **append()**. In order to get a list of integers again, you need to use **integer division** (double division slash).

USING THE LIST INDEX

In the following example, you go through the list using the **list index** and **change the list elements** without creating a new list. Display the sound graphically before and after the change.



```
from soundsystem import *
from gpanel import *
```

```
samples = getWavMono("mysound.wav")

makeGPanel(0, len(samples), -33000, 33000)
for i in range(len(samples)):
    if i == 0:
        move(i, samples[i] + 10000)
    else:
        draw(i, samples[i] + 10000)

for i in range(len(samples)):
    samples[i] = samples[i] // 4

for i in range(len(samples)):
    if i == 0:
        move(i, samples[i] - 10000)
    else:
        draw(i, samples[i] - 10000)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

People often use the variable name *i* as a list index. If you run through a loop block using a for structure

```
for i in range(10):
```

i is also called a **stepper**.

GENERATING SOUNDS

It is exciting to create your own sounds, not by loading a sound list from a sound file, but rather by creating the list elements yourself. To make a "rectangular wave sound" you repeatedly store the value 5000 in the list, for a certain index range, and subsequently -5000 for the same index range.

```
from soundsystem import *

samples = []
for i in range(4 * 5000):
   value = 5000
   if i % 10 == 0:
       value = -value
   samples.append(value)

openMonoPlayer(samples, 5000)
play()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The sampling rate of 10000 Hz corresponds to a sound sample every 0.1 ms. We want to change the sign (-/+) always after 10 values, in other words, every 1 ms. This corresponds to a rectangular wave period of 2 ms, which yields a sound of 500 Hz. We use the modulo operator %, which returns the remainder of the integer division. The condition **i % 10 == 0** is then true for i = 0, 10, 20, 30, etc.

EXERCISES

1. Use the list operation *reverse()* to play a sound backwards, e.g. a spoken text.

2. With the slice notation list[start: end] you can create lists that contain only the elements with the index *start* to *end* (without the last element). Using this, remove a part of your present sound.

3. Load a sound clip and determine the maximum amplitude value. Write it in the title bar of the GPanel and display the sound graphically. Now increase all of the sound samples, so that the maximum amplitude value amounts to 32767 (the maximum volume) and display the clip again. (This is an important function of most sound editors, called normalizing).

4*. Create a sine wave of about 500 Hz with a sampling rate of 10000 Hz using the sine function *math.sin(x)*, which always restarts after x = 2π = 6.28. To get access to the sine function you have to include *import math*.

5*. Superimpose two sine waves with neighboring frequencies. What do you notice while listening?

4.3 RECORDING SOUNDS

INTRODUCTION

You can also record and save sounds with the sound system. To get started, you have to connect the sound card input with an external source, e.g. a microphone or a playback device. Laptops usually have a built-in microphone.

PROGRAMMING CONCEPTS: *Blocking and non-blocking function*

SOUND RECORDER

Before recording, call **openMonoRecorder()** to prepare the recording system. You have to specify the sampling rate as a parameter. You can start recording with **capture()**. This function is non-blocking and returns immediately. You will have to call **stopCapture()** later to end the recording. The recorded sound samples are copied into a list that you can get with **getCapturedSound().** Here you make a recording of 5 seconds duration and then play the sound.

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

openMonoPlayer(sound, 22050)
play()
```

MEMO

A command like **capture()**,which triggers an action and immediately returns, is also called a **non-blocking function**. With such functions you are able to control tasks from your ongoing program, while these tasks are executed in the background. For instance, you can abort them.

ILLUSTRATING RECORDED SOUND

Of course, we are often interested in the graphical representation of the recorded sound. You should already know how you can do this with the GPanel.

The adjacent graphics shows the recording of the words
"one two three for five six seven eight nine ten".

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

from gpanel import *
makeGPanel(0, len(sound), -33000, 33000)
for i in range(len(sound)):
    draw(i, sound[i])
```

MEMO

You can obtain the number of samples from the length of the sound list. For the graphical representation, simply use a *for* structure. Play around for a while with different recorded sounds and think about whether you understand the sound curve.

SAVING WAV FILES

You can also save the recorded sound as a WAV file with **writeWavFile()**.

```
from soundsystem import *

openMonoRecorder(22050)
print("Recording...");
capture()
delay(5000)
stopCapture()
print("Stopped");
sound = getCapturedSound()

writeWavFile(sound, "mysound.wav")
```

MEMO

After saving, you can listen to the sound file with either *Python* or with any sound player that is installed on your computer.

EXERCISES

1. Record individual words

2. Put these words together in a sentence..

3*. Let the computer speak in single digits a phone number given as a text.

4.4 SPEECH SYNTHESIS

INTRODUCTION

In speech synthesis, a human voice is generated by the computer. A text-to-speech system (TTS) converts written text into a speech output. The automatic generation of human language is complicated, but it has made a lot of progress in recent years. Compared to the playback of pre-made voice recordings, TTS has the advantage of being very flexible and able to speak any text. Speech synthesis is a part of computational linguistics. Therefore, a close collaboration between linguists and computer scientists is necessary in the development of a TTS.

The speech synthesis software used in *TigerJython* is called *MaryTTS* and was developed at the Department of Computational Linguistics and Phonetics of the University of Saarland in Germany.

The system uses large library files that you download separately **here** and then unzip. In the same directory as *tigerjython2.jar*, create the subdirectory *Lib* (only if it does not already exist) and copy the unzipped files into it.

PROGRAMMING CONCEPTS: *Speech synthesis, artificial speech, text-to-speech system*

SPEAKING A TEXT IN 4 LANGUAGES

In this release, MaryTTS provides you with different voices speaking German, English, French and Italien.. You can choose the voice with **selectVoice()**. After that you can call the function **generateVoice()** by passing it the text to be spoken. It will return a list with the generated sound samples that you can play back with a sound player.

```
from soundsystem import *

initTTS()

selectVoice("german-man")
#selectVoice("german-woman")
#selectVoice("english-man")
#selectVoice("english-woman")
#selectVoice("french-woman")
#selectVoice("french-man")
#selectVoice("italian-woman")

text = "Danke dass du mir eine Sprache gibst. Viel Spass beim Programmieren"
#text = "Thank you to give me a voice. Enjoy programming"
#text = "Merci pour me donner une voix. Profitez de la programmation"
#text = "Grazie che tu mi dia una lingua. Godere della programmazione"
voice = generateVoice(text)
openSoundPlayer(voice)
play()
```

MEMO

You can change the commented lines to let the program speak the text using the different voices. You first always have to call **initTTS()** in order to prepare the speech synthesis software.

You could also pass the function *initTTS()* a path to the directory containing the *MaryTTS* data files as a parameter. By default it is the subdirectory *Lib*.

ANNOUNCING TODAY'S DATE AND THE CURRENT TIME

There are numerous applications of speech synthesis. People with visual impairments can have texts read aloud to them, and navigation systems or train station or train announcements often use synthetically generated voices.

Many interactive computer games also use artificially generated voices.
Your program determines the current time from the computer system, and it reads it out loud with a German or an English speaking voice.

```
from soundsystem import *
import datetime

language = "german"
#language = "english"

initTTS()
if language == "german":
    selectVoice("german-woman")
    month = ["Januar", "Februar", "März", "April", "Mai",
        "Juni", "Juli", "August", "September", "Oktober",
        "November", "Dezember"]
if language == "english":
    selectVoice("english-man")
    month = ["January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December"]

now = datetime.datetime.now()

if language == "german":
    text = "Heute ist der " + str(now.day) + ". " \
        + month[now.month - 1] + " " + str(now.year) + ".\n" \
        + "Die genaue Zeit ist " + str(now.hour) + " Uhr " + str(now.minute)
if language == "english":
    text = "Today we have  " + month[now.month - 1] + " "  \
        + str(now.day) + ", "+ str(now.year) + ".\n" \
        + "The time is " + str(now.hour) + " hours " + str(now.minute)
        + " minutes."

print text
voice = generateVoice(text)
openSoundPlayer(voice)
play()
```

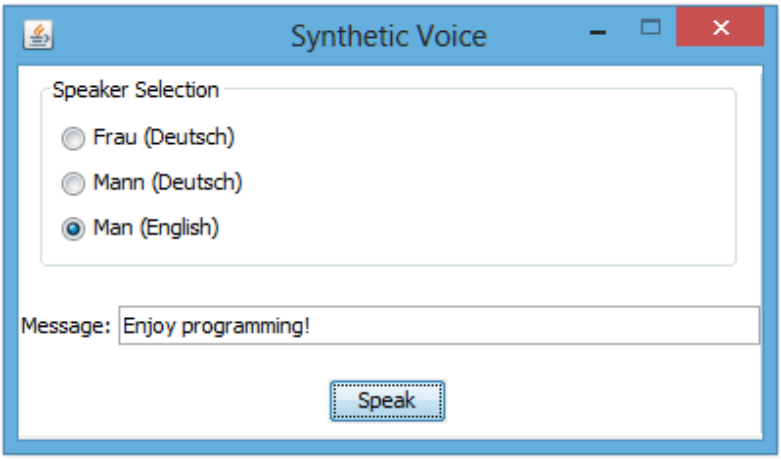
MEMO

By selecting the commented lines, you can decide between the German or the English speaker. The class **datetime.datetime.now()** provides you with information about the current date and the current time, via its attributes *year, month, day, hour, minute, second, microsecond*. As you can see, you can use the backslash as a line extension in the definition of long strings.

CREATING YOUR OWN GRAPHICAL USER INTERFACE

As you have already learned in chapter 3.13 it is quite easy to create a simple dialog window based on TigerJython's EntryDialog class. As usual in many programming environments the classic controls like text fields, push, check and radio buttons, as well as sliders are modeled by software objects. These objects appear in a surrounding rectangular pane and the dialog remains open while the program continues (such a dialog is called a modeless dialog). For a comprehensive information please consult the APLU documentation.

Your program opens a modeless dialog where you select the speaker using radio buttons. When clicking the confirmation button, the text in the text field is read by a synthetic voice.




```
from soundsystem import *
from entrydialog import *

speaker1 = RadioEntry("Mann (Deutsch)")
speaker1.setValue(True)
speaker2 = RadioEntry("Man (English)")
speaker3 = RadioEntry("Homme (Français)")
speaker4 = RadioEntry("Donna (Italiano)")
pane1 = EntryPane("Speaker Selection",
                  speaker1, speaker2, speaker3, speaker4)
textEntry = StringEntry("Message:", "Viel Spass am Programmieren")
pane2 = EntryPane(textEntry)
okButton = ButtonEntry("Speak")
pane3 = EntryPane(okButton)
dlg = EntryDialog(pane1, pane2, pane3)
dlg.setTitle("Synthetic Voice")

initTTS()

while not dlg.isDisposed():
    if speaker1.isTouched():
        textEntry.setValue("Viel Spass am Programmieren")
    elif speaker2.isTouched():
        textEntry.setValue("Enjoy programming")
    elif speaker3.isTouched():
        textEntry.setValue("Profitez de la programmation")
    elif speaker4.isTouched():
        textEntry.setValue("Godere della programmazione")

    if okButton.isTouched():
        if speaker1.getValue():
            selectVoice("german-man")
            text = textEntry.getValue()
        elif speaker2.getValue():
            selectVoice("english-man")
            text = textEntry.getValue()
        elif speaker3.getValue():
            selectVoice("french-man")
            text = textEntry.getValue()
        elif speaker4.getValue():
            selectVoice("italian-woman")
            text = textEntry.getValue()
        if text != "":
            voice = generateVoice(text)
            openSoundPlayer(voice)
            play()
```

MEMO

The while loop executes until the dialog is closed with the title bar's close button. You check with **isTouched()** in every cycle, if the confirmation button was clicked since the last call of this function. In this case you get the current values of the GUI elements by calling **getValue()**and transform the text in the text field to a voice like in the preceding examples.

It is a bit dangerous to go through such "narrow" loops, because you waste lot of processing time for nothing other than just a check whether the button was pressed. However, when you call *isTouched()* the program will automatically stop for a short time (1ms) so that the throughput is slightly slowed down.

EXERCISES

1. Find or write a short poem as a text file, for example:

   Advice To A Son by Ernest Hemingway.

   Never trust a white man,
   Never kill a Jew,
   Never sign a contract,
   Never rent a pew.
   Don't enlist in armies;
   Nor marry many wives;
   Never write for magazines;
   Never scratch your hives.
   Always put paper on the seat,
   Don't believe in wars,
   Keep yourself both clean and neat,
   Never marry whores.
   Never pay a blackmailer,
   Never go to law,
   Never trust a publisher,
   Or you'll sleep on straw.
   All your friends will leave you
   All your friends will die
   So lead a clean and wholesome life
   And join them in the sky.

   Ernest Hemingway (**Download)**

   With the line *text = open("poem.txt", "r").read()* you can read the text from the text file *sorcery.txt,* in the same directory as your program, as string. Let the text be read by the English voice.

2. Define the function *fac(n)* either iteratively or recursively, which returns the factorial

   n! = 1 * 2 * ... *n

   Your program should ask you for a number between 0 and 10 using *readInt()* and also speak the question out loud. It then calculates the factorial n! of the entered number and outputs the result as spoken text.

4.5 ACOUSTIC EXPERIMENTS

INTRODUCTION

You can also use the computer in place of an experimental system, for example you could investigate human hearing using the sound system. This is not only cheaper, but it also gives you a huge amount of flexibility, especially when you perform the experiments with a self-written program.

PROGRAMMING CONCEPTS: *Concert pitch, beating, scale*

TUNING A MUSICAL INSTRUMENT, BEATING

The hearing cannot distinguish two tones with almost the same frequencies when they are played separately. However, if they are played simultaneously this results in a rise and fall in volume which is very well audible. In order to experience this yourself, make your program play a standard concert pitch A (440 Hz) for 5 seconds and later for the same period an only 1 Hz higher pitch. There is no noticeable difference. When playing both tones at together you can clearly hear the beating phenomenon.

```
import time

playTone(440, 5000)
time.sleep(2)
playTone(441, 5000)
time.sleep(2)
playTone(440, 20000, block = False)
playTone(441, 20000)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The global function *playTone()* has different parameter variations that you can read about in the *TigerJython Help* file under *APLU Documentation* (coordinate graphic). Here you use the parameter *block,* with which you can define whether the function blocks until the tone is done being played, or whether it returns immediately after starting to play. You have to use the non-blocking variant to play multiple sounds simultaneously.

To tune instruments in an orchestra, but also to tune a single instrument (string instrument, piano, etc.), two notes are played at the same time while paying attention to the beating.

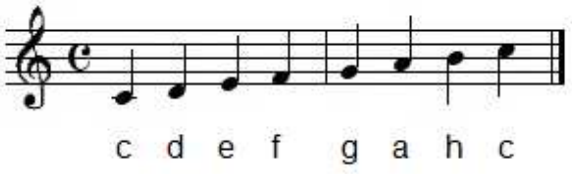
SCALES

The well-tempered musical scale is based on a standard concert pitch with the frequency 440 Hz and divides the octave (frequency ratio 2) in 12 semitones with the same frequency ratio *r*. Thus gives:

$$r^{12} = 2 \quad \text{or} \quad r = \sqrt[12]{2} \approx 1.0594630943$$

You can easily play the C major scale with this, which according to the notation, consists of

whole and half steps. The concert pitch corresponds to the note a.



In the just or natural scale the tone frequencies are formed by multiplication with simple ratios starting from the root tone. The ratios for the 8 tones of an octave are:

$$1, \frac{9}{8}, \frac{5}{4}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, \frac{15}{8}, 2$$

or as a series of numbers, they are: 24, 27, 30, 32, 36, 40, 45, 48. To play these, you can save the frequencies in a list and call *playTone()*. Once you have played both scales individually, you can listen to the two differently tuned instruments playing the scale together. As you will notice, it sounds really bad.

```
r = 2**(1/12)
a = 440
c = a / r**9

scale_temp = [c, c * r**2, c * r**4, c * r**5, c * r**7,
              c * r**9, c * r**11, 2 * c]
scale_pure = [c, c * 9/8, c * 5/4, c * 4/3, c * 3/2 ,
              c * 5/3, c * 15/8, 2 * c]

playTone(scale_temp, 1000)
playTone(scale_pure, 1000)

playTone(scale_temp, 1000, block = False)
playTone(scale_pure, 1000)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In the well-tempered scale, neighboring semitones always have the same frequency ratio (so, not equal frequency differences!). The advantage of the well-tempered scale over just intonation is that the frequency ratios are always the same for all keys (C major, D major, etc.). [**more...**]

PLAYING MELODIES

You can also play a simple melody for fun using *playTone()*. For subsequent tones of equal length, use tuples with a pitch and speed indication, and put them into a list. Finally, it is also possible to choose a musical instrument. In this example you probably recognize a children's tune. Which one is it?

```
v = 250
playTone([("cdef", v), ("gg", 2*v), ("aaaa", v//2), ("g", 2*v),
          ("aaaa", v//2), ("g", 2*v), ("ffff", v), ("ee", 2*v),
          ("dddd", v), ("c", 2*v)], instrument="harp")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

It is really amazing how easily you can play a melody using *playTone()*. However, compared to a real musical instrument, it sounds very synthetic.

EXERCISES

1. You can write down a song as a list of tone frequencies and then play it with a *for* loop:

```
melody = [262, 444, 349, 349, 392, 330, 262, 466, 440, 392, 392, 349]
v = 200
for f in melody:
    playTone(f, v)
```

   a. Do you know this song? Play it back a bit slower.

   b. Play the song one octave higher.

   c. For your singing class, the first version is too low and the second is too high. Transpose the melody so that it starts with g' instead of c'.

2. Play the chord c'', e'', g'' (third, fifth) for twenty seconds with the well-tempered intonation. (For this, you can use *playTone()* giving it letters for the tones.) Now play the same chord, but with just intonation. What do you notice?

Sound Documentation

Sound

Function	Action
playTone(freq)	plays tone mit given frequency (in Hz) and duration 1000 ms (blocking function)
playTone(freq, blocking=False)	same, but not-blocking function, used to play several tones at (about) the same time
playTone(freq, duration)	plays tone with given frequency and given duration (in ms)
playTone([f1, f2, ...])	plays several tones in a sequence with given frequency and duration 1000 ms
playTone([(f1, d1), (f2, d2), ...])	plays serveral tones in a sequence with given frequency and given duration
playTone([("c", 700), ("e", 1500), ...])	plays serveral tones in a sequence with given (Helmholtz) pitch naming and duration. Supported are: great octave, one-line to three-line octave (range C, C# up to h'''
playTone([("c", 700), ("e", 1500), ...], instrument = "piano")	same, but selects instrument type. Supported are: piano, guitar, harp, trumpet, xylophone, organ, violin, panflute, bird, seashore, ... (see MIDI specifications)
playTone([("c", 700), ("e", 1500), ...], instrument = "piano", volume=10)	same, but selects sound volume (0..100)

Module import: from soundsystem import *

Playback:

getWavMono(filename)	provides lists of samples for the specified monaural sound file. "wav/xxx.wav" also loads from the _wav directory in *tigerjython2.jar*
getWavStereo(filename)	provides lists of samples for the specified binaural sound file. "wav/xxx.wav" also loads from the _wav directory in *tigerjython2.jar*
getWavInfo(file)	provides a string with information about the sample rate, etc.
openSoundPlayer(filename)	opens a sound player with the specified sound file. Afterwards, it can be played with the following player functions
openMonoPlayer(filename)	opens a monaural sound player with the specified sound file. It can also handle binaural files (average of both channels)
openStereoPlayer(filename)	opens a binaural sound player with the specified sound files. It can also handle monaural files (both channels are identical)
openSoundPlayerMP3(filename)	just like openSoundPlayer(), but for MP3 files
openMonoPlayerMP3(filename)	just like openMonoPlayer(), but for MP3 files
openStereoPlayerMP3(filename)	just like openStereoPlayer(), but for MP3 files
play()	plays the sound from the current position and returns immediately
blockingPlay()	plays the sound from the current position and then waits until the playing has finished
advanceFrames(n)	fast forwards the current position by a given number of samples
advanceTime(t)	fast forwards the current position by a specified time
getCurrentPos()	returns the current position
getCurrentTime()	returns the current playing time
rewindFrames(n)	rewinds the current position by the specified number of samples
rewindTime(t)	rewinds the current position by the specified time
stop()	stops playing and resets the current playhead to the beginning
setVolume(v)	sets the volume (v = 0...100)
isPlaying()	returns True if the clip has not yet finished playing

mute(bool)	turns to True when muted, and False when audible
playLoop()	loops, and therefore plays the clip endlessly
replay()	replays the clip once
delay(time)	delays the program (in milliseconds)

Recording and Saving:

openMonoRecorder()	opens a monaural sound recorder
openStereoRecorder()	opens a binaural sound recorder
capture()	begins recording
stopCapture()	stops recording
getCapturedBytes()	returns the recorded samples byte-by-byte
getCapturedSound()	returns the recorded samples as integer list values (binaural: channels alternate)
writeWavFile(samples, filename)	writes the samples into a WAV file

Fast Fourier Transform (FFT):

fft(samples, n)	transforms the first *n* values of the specified list of samples (floats). Returns a list with n // 2 equidistant spectral values (floats). At a sampling rate of *fs* these range from 0 to fs/2 at a distance fs/n (resolution)
sine(A, f, t)	creates a sine wave with amplitude A and frequency *f* (phase 0) for each float value *t*
square(A, f, t)	creates a square wave with amplitude A and frequency *f* (phase 0) for each float value *t*
sawtooth(A, f, t)	creates a sawtooth wave with amplitude A and frequency *f* (phase 0) for each float value *t*
triangle(A, f, t)	creates a triangle wave with amplitude A and frequency *f* (phase 0) for each float value *t*
chirp(A, f, t)	creates a sine wave with amplitude A and a frequency that increases linearly with time (initial value *f*) for each float value *t*

chapter five

ROBOTICS

Learning Objectives

- You can describe what a robot is and you know some of their possible applications.
- You know the difference between an autonomous and a remotely controlled robot, and you know why robots are simulated.
- You can control EV3 or NXT robots with a *Python* program.
- You can explain, using a few examples, what a learning robot is. You also understand the difference between the *teach* and *execute* modes.
- You know the principles of a control system and can list some examples of controls.
- You can capture sensor values in a program using polling and events.

"Will robots inherit the earth? Yes, but they will be our children."

but:

"No computer has ever been designed that is ever aware of what it's doing; but most of the time, we aren't either."

Marvin Minsky, AI Researcher at MIT

5.1 REAL AND SIMULATION MODE

INTRODUCTION

A robot is usually understood as a computer-controlled machine that can perform an activity, previously done by humans. If the machine can also detect the surrounding environment with the help of cameras and sensors and can then react appropriately with actuators (motors, valves, speech synthesizers, etc.), we speak of an **intelligent system.** If the behavior of such a system is human-like, we speak of an **android**. .

A typical example of this is the movie robot *WALL-E* who has his own consciousness, so much that he is looking for spare parts for himself, as well as special objects that catch his interest, which he treasures in a collection. He is also able to solve a Rubik's cube, which is definitely seen as a sign of intelligence.

Artificial Intelligence (AI) deals with the interesting question of exactly how a computer system can be described as "*intelligent*". In order to answer this question, we must first define what is meant by an "intelligent" machine. One possible approach to the definition of intelligence is the **Turing Test**.

Here you will be concerned with more simple questions and you will learn to handle a simple robot equipped with touch, light, sound, infrared, and ultrasound sensors, and also with two wheels driven by electric motors, which allow it to move forwards and backwards, and even rotate.

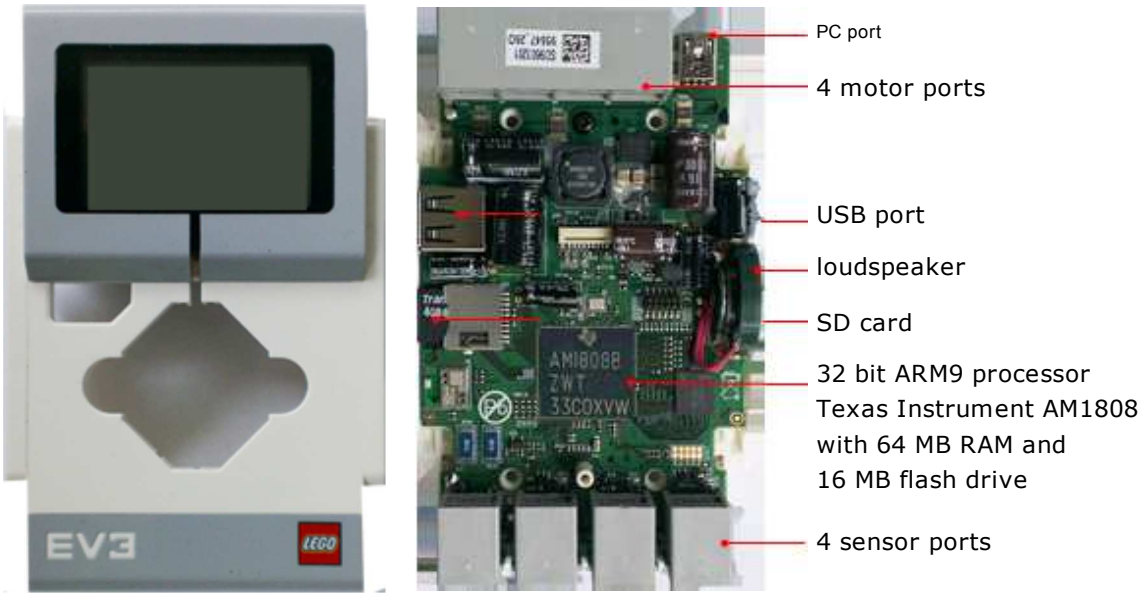
The motors and sensors are controlled by the built-in computer, which is why people also call robots an **embedded system**. If it often consists in a simple computer chip, it is called a **microprocessor** (or a **microcontroller**). Nowadays, embedded systems play an extremely important role and you can find them in many everyday devices, for instance in smartphones. Surprisingly, most coffee machines, washing machines, televisions, cameras, and other similar electronics are also embedded systems. In a modern car, there are up to 100 microcontrollers that act as embedded systems in places such as the engine control or the anti-blocking system. Therefore, you should be aware that you are also getting to know many general principles for embedded systems while learning about robots.

If the built-in processor runs a stand-alone program in order to control the robot, we speak of an **autonomous** robot. The built-in processor can also simply send the data collected from the sensors over a data communication canal to an external computer, and then obtain control commands from this computer. In this case, we speak of an **remotely controlled** robot. Finally, a robot can also be **simulated,** which usually means that the sensors, motors, etc. are depicted as software objects. A class construction then corresponds to the real-world assembly of robot components. In general practice, robots are usually first simulated on the computer since this can create a behavior to be studied with little effort and without any risks to the environment.

With the world famous robotics kit by **LEGO Mindstorms** you can learn the important aspects of robotics in a playful way. The kit consists of a microprocessor-controlled **brick** and a variety of

components used to construct different robot models. The brick has gone through several stages of development: earlier it was called RCX, then NXT, and more recently EV3.

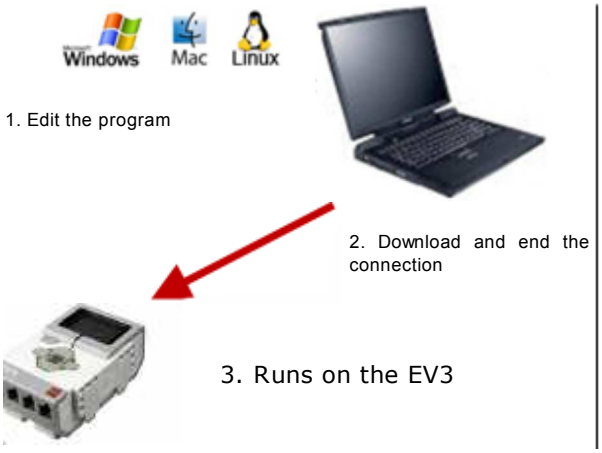
The EV3 brick is an embedded system with motors and sensors controlled by a modern ARM processor. If you open it, its electronic components will be visible.



Once you turn on the brick, a firmware starts on the microcontroller (or with EV3 the Linux operating system) and a simple menu appears on the display. With this, you can already run programs stored on the brick, in autonomous mode. For the external control mode on the EV3, you have to start a helper program (BrickGate) which interprets commands that are received through a Bluetooth connection, for example the command to turn on a motor in a certain rotation direction or to report back the measured value of a senor. With the NXT, this program is a part of the firmware.

As with all embedded systems in robotics, you need an external computer on which you develop the robot programs. In autonomous mode the program is downloaded onto the brick, and in external control mode it runs on the PC.

Autonomous mode



External control mode

PROGRAMMING CONCEPTS: *Robots, androids, artificial intelligence, embedded systems, microprocessors, microcontrollers, blocking/non-blocking methods*

PREPARATIONS

With *TigerJython* you can simulate the robot (**simulation mode**) or using autonomous or external control modes (**real mode**). You thereby use different class libraries which, however, offer the same programming interface (Application Programming Interface, API) so that the programs are practically identical for all modes. The only things that have to be adjusted are imports and possibly some timings used in the program.

Simulation mode:
If you do not have an NXT or an EV3 available to you, you can nonetheless work through the topic in simulation mode. The images required for the simulation are already included in the distribution of *TigerJython*.

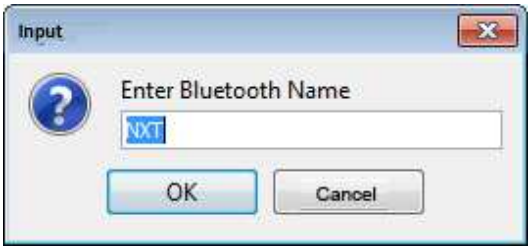
Real mode:

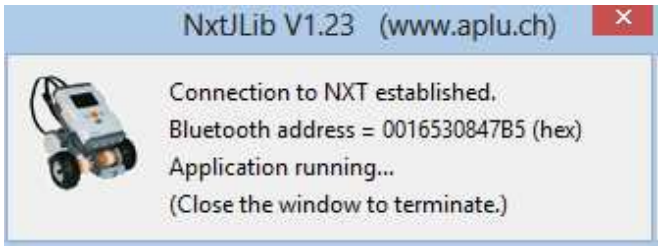
Most examples will use the basic model of the LEGO Mindstorm NXT or EV3 robot, which moves with two motors and can be equipped with various sensors. As long as you do not change the basic functionality, you can also use your own deviated model. Since the robot communicates through Bluetooth, your PC must be Bluetooth compatible and enabled. Moreover you have to "pair" the brick with your computer.

Using the LEGO NXT:
Make sure that the Java firmware leJOS is installed onto the LEGO NXT. Here you find instructions how to proceed: **http://www.legorobotik.ch/lejosfirmware_en**

To pair it with the computer, you proceed as you would with other external Bluetooth devices such as smartphones, Bluetooth handsets, printers, etc.

With Python, you can use the NXT only in its external control mode. For this, you write a regular Python program in *TigerJython* using the module *ch.aplu.nxt* and then hit the green Run button to start. First you will be asked for the Bluetooth name and then the connection with the brick will be established. During the program execution, a window with the connection information remains open. If you close this window, the connection to the NXT is interrupted.





If there are several LEGO NXT's in the room, the names for each must be different so that there are no conflicts. You can find a tool for changing the name **here**. You can also use the Bluetooth address instead of the Bluetooth name, which you can figure out with the help of different tools (one place to find it is shown above: it is written out in the connection dialog each time that a connection is established).

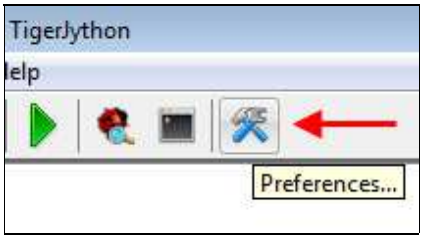
The BlueCove library is necessary for Bluetooth communication. Download the files **here** and unzip them in the subdirectory *Lib* of the directory in which *tigerjython2.jar* is located.

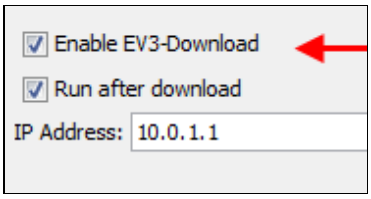
Using the LEGO EV3:
A Linux operating system runs on the EV3 that boots in conjunction with the leJOS software, which is located on the SD card. You can find a detailed guide on how to create the SD card **here**. If you remove the SD card you can use the EV3 in its original state. If the EV3 is started using leJOS you communicate with it through a Bluetooth PAN connection. To do this, you have to pair the PC with the brick and specify it as a network access point. You can find instructions **here**.

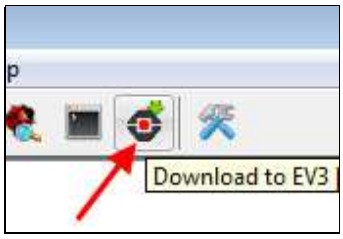
After booting the EV3 with leJOS and successfully connecting it via Bluetooth PAN, you need to start the ***BrickGate*** server on the brick, which you can find in the menu "programs".

To use the EV3 in autonomous mode, check both boxes on the *libraries* tab in the settings of

TigerJython, activate EV3 download and *run after download*. You will then find an additional EV3 icon on the toolbar.







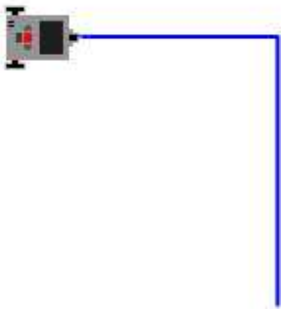
For the external control mode, you click on the green Run button as usual. Just as with the NXT, you will first be asked for the Bluetooth name and then a window will open with the connection information. If you want to run the same program autonomously, simply click on the EV3 button. The *Python* script is then downloaded onto the EV3 and executed there. Its name also appears on the display of the EV3 and it can always be executed again with the Enter button, even without a connection to the PC.

In the programs we assume that for the EV3 you are using the new motors and sensors from the EV3 product line. The EV3 color sensor also serves as a light sensor. However, the old NXT motors and sensors are still supported for the EV3. You simply have to put "Nxt" everywhere before the class name, i.e. NxtMotor, NxtGear, NxtTouchSensor, etc.

MOVING FORWARDS AND BACKWARDS, TURNING

In your first robot program, the robot should move for certain times, which are hard coded in the program. The robot library is object-oriented and depicts reality by a model. So, when in reality you pick up the LEGO brick when constructing the robot, in the software you create an instance of the class *LegoRobot()* with **robot = LegoRobot()**. Next you take two motors and put them together to a gear, which you can express as **gear = Gear()** in the software. Then you connect the gear motors to the motor ports A and B, which you formulate as **addPart()** in the software.

With the command *gear.forward()* you turn on both engines simultaneously with the same rotational speed and the robot moves straight ahead. **This state of movement will remain the same until you undertake something else.** However, the call returns immediately and your program continues onwards with the next instruction (this is called a **non-blocking method**). Therefore, your program has to ensure that the robot does something else after a certain amount of time. To do this, you can tell the program to wait using **Tools.delay()** and then change or stop the movement with another command.



Once you send a movement command to the robot, the current state is ended and replaced by the new state. At the end of the program you should always call the method **exit()**. Once called, all motors are stopped and in the real mode the Bluetooth connection is also interrupted, which is necessary for the next program to start successfully. (If the program does not end correctly you might have to turn the brick off and on again.)

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)

gear.forward()
Tools.delay(2000)
gear.left()
Tools.delay(545)
```

```
gear.forward();
Tools.delay(2000)
robot.exit()
```

MEMO

A Gear has two motors. Instead of controlling the motors individually, you can use commands that affect both engines simultaneously.

The class libraries for both the simulation and the real mode are designed so that their programs are almost identical. You can first develop your program in the simulation mode and then with a few adaptations you can execute it with the real robot.

You can use the EV3 autonomously or externally controlled. In both cases the BrickGate program must be started on the EV3, which receives and appropriately interprets the commands sent from *Python*. Since no errors are displayed while running in autonomous mode, you should always first test the program in external control mode (green button) and only then download it to the brick using the EV3 button and execute it there.

MOVING WITH BLOCKING METHODS

Instead of moving the robot forward with the command *forward()* and then telling the program to wait 2000 ms with *delay(2000)*, you can use the blocking method **forward(2000)** which also moves the robot forward, but only returns after 2000 ms. There are also blocking variants for **left()** and *right()*.

You can simplify the previous program slightly with blocking methods.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.forward(2000)
gear.left(545)
gear.forward(2000)
robot.exit()
```

MEMO

You must distinguish between blocking and non-blocking methods. Non-blocking commands cause the robot to change state and return immediately. If you call a blocking method, the program freezes a certen time interval, i.e. the next statement is only executed when the time interval has expired.

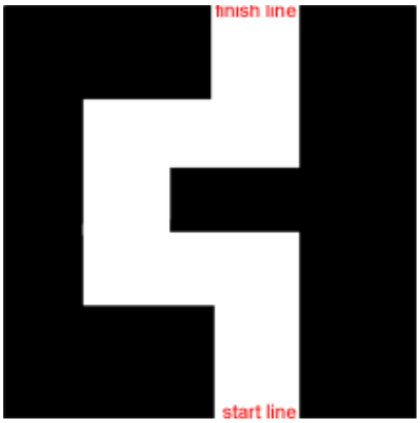
At first glance, it might seem easier to always use blocking methods. But with them you get a major disadvantage. While your program is blocked you cannot execute any other actions, so for example, you could not read any sensor values! If the program hangs during execution, in external control mode you can cancel it by closing the connection information window. In autonomous mode in the case of an emergency, you can simultaneously press the two buttons DOWN+ENTER .

EXERCISES

1. Write a program that makes your robot trace a square, using blocking methods.

2. Write a program using non-blocking methods so that the robot moves along half-circle curves.

3. Create a course with some objects and write a corresponding program so that the robot moves from the start to the finish line.



For simulation mode, you can use the background image *bg.gif* located in the subdirectory *sprites,* by displaying it with *RobotContext.useBackground()*.

Using *RobotContext.setStartPosition()* you can set the robot to a specific location at the start of the program. (window coordinates are from 0 to 500, 0 is at the top left corner).

```
RobotContext.setStartPosition(200, 455)
RobotContext.useBackground("sprites/bg.gif")
```

You can also create your own image (the image size should be 501x501).

ADDITIONAL MATERIAL

INFRARED REMOTE CONTROL

A versatile infrared sensor comes with the EV3 robot that can be used in many ways. It is already included in the LEGO Home set (including the remote control box), but it must be purchased separately if you have the Education set. You can use the IRSensor in one of the three following ways:

Class	**Metrics**
IRSeekSensor	distance and direction to the IR source of the remote control
IRRemoteSensor	pressed buttons of the remote control
IRDistanceSensor	distance to a reflective target

The use of the remote control is fun and motivating for your first "tentative steps" with the robot. As opposed to a predetermined remote control program, you can set the actions that which are triggered when pressing the remote control through simple *Python* programming.

You decide to use the following actions in your program:

Remote control button	Action
Left-Up	moves forward on a left bend
Right-Up	moves forward on a right bend
Left-Down+Right-Up	moves straight forward
Left-Down	stops
Right-Down	ends program

```
from ev3robot import *

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
irs = IRRemoteSensor(SensorPort.S1)
robot.addPart(irs)
isRunning = True

while not robot.isEscapeHit() and isRunning:
    command = irs.getCommand()
    if command == 1:
        gear.leftArc(0.2)
    if command == 3:
        gear.rightArc(0.2)
    if command == 5:
        gear.forward()
    if command == 2:
        gear.stop()
    if command == 4:
        isRunning = False
robot.exit()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The methods *isEscapeHit(), isEnterHit(), isDownHit(), isUpHit(), isLeftHit(), isRightHit()* return *True*, if you click the corresponding buttons on the EV3 in autonomous mode.

However, in external control mode, they pertain to the keyboard keys ESCAPE, ENTER, CURSOR-DOWN, CURSOR-UP, CURSOR-LEFT, CURSOR-RIGHT. For this, the connection information window must be active (click in it with the mouse to activate it).

5.2 INTELLIGENT ROBOTS

INTRODUCTION

Robots that can find their way in a changing environment have many potential applications, for example as flying objects, in underwater exploration, and in the examination of sewer systems. Here you will learn step by step how you can build a moving robot that is able to orient itself in a changing environment

PROGRAMMING CONCEPTS: *Externally controlled, autonomous, self-learning robot, teach mode, execution mode, event loop*

THE ROBOT KNOWS THE WAY

In the simplest case, a robot should be able to find a path in a very special canal that consists of elements of the same length arranged orthogonally.

Information about the constant length of the canal elements and whether they consist of left or right curves is hardcoded ("wired") in the program.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)

moveTime = 3200
turnTime = 545

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.forward(moveTime)
gear.left(turnTime)
gear.forward(moveTime)
gear.right(turnTime)
gear.forward(moveTime)
gear.right(turnTime)
gear.forward(moveTime)
gear.left(turnTime)
gear.forward(moveTime)
robot.exit()
```

MEMO

You have to figure out **moveTime** and **turnTime** through a series of experiments and then adjust accordingly. Naturally, they correlate to the speed of the robot. In reality, you would probably rather specify the route to be traversed and the rotation angles instead of the times.

ROBOT CONTROLLED BY A HUMAN

The robot knows the constant lengths of the canal elements, but its turning movements are controlled by a human. However, the robot is not capable of learning and it cannot remember the path, so it remains "stupid". To control the robot in both the simulation and external control mode, you use the left and right cursor keys of the keyboard, and in autonomous mode you use the corresponding LEFT and RIGHT buttons. With the methods *isLeftHit()* and *isRightHit()* you can ask whether the keys or the buttons were pressed and again released. Use the escape key or ESCAPE button to exit the program.

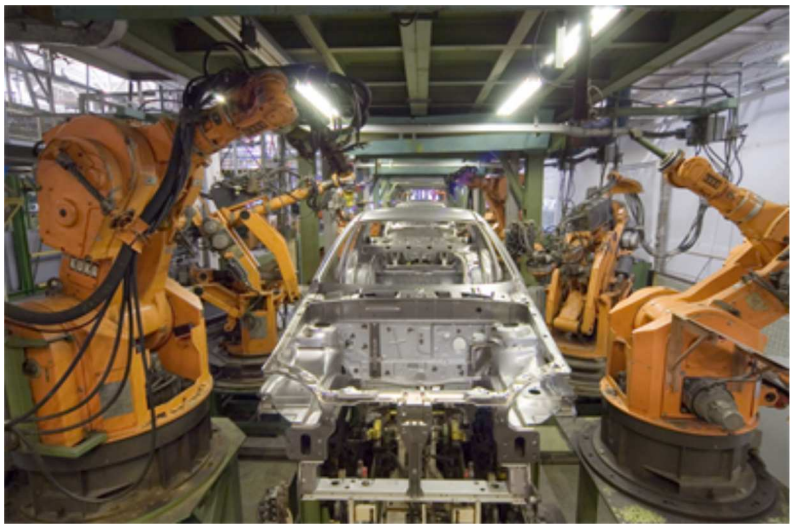
```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)

moveTime = 3200
turnTime = 545

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
gear.forward(moveTime)

while not robot.isEscapeHit():
    if robot.isLeftHit():
        gear.left(turnTime)
        gear.forward(moveTime)
    if robot.isRightHit():
        gear.right(turnTime)
        gear.forward(moveTime)
robot.exit()
```

MEMO

In this case, it makes less sense to use the autonomous mode since you actually want to remote control the robot. You can also use the infrared remote control for this with the EV3 instead of using the keyboard (see Additional Material at the end of this chapter).

THE ROBOT LEARNS IN TEACH MODE

Computer-aided systems, whose behavior is not hardcoded and who can therefore later adapt their behavior to an environment, are called **adaptive systems**. These are therefore **capable of learning,** in a way. Industrial robots are "trained" by specialists in a "**teach mode**", for instance which arm movements are to be carried out. In most cases, the operator uses an input system similar to a remote control. The robot is successively moved to the desired positions and the respective state is stored. In "**execution mode**" the robot runs through the stored states independently (and with a higher speed).



As before, your canal robot knows the constant length of the canal elements, but its turning movements are controlled by a human. However, the robot is now able to learn, so it can memorize the path and independently run through it any number of times.

It is often useful to imagine that in every moment, a robot is in a particular **state**. The states are typically labeled with meaningful words and stored as a string. You assume the following states: the robot is stopped, moving forward, turning left, or right, and you call them: STOPPED, FORWARD, LEFT, RIGHT. [**more...**]

Instead of constantly querying the keys or buttons, here you use a more elegant event programming model with registered callback functions, which are, independently of the currently running program, always called automatically when an event occurs.

The main program, in an endless loop, is engaged in performing the corresponding actions in each state. The state change takes place in the callback *onButtonHit()*.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)
RobotContext.showStatusBar(30)

def onButtonHit(buttonID):
    global state
    if buttonID == BrickButton.ID_LEFT:
        state = "LEFT"
    elif buttonID == BrickButton.ID_RIGHT:
        state = "RIGHT"
    elif buttonID == BrickButton.ID_ENTER:
        state = "RUN"

moveTime = 3200
turnTime = 545
memory = []
robot = LegoRobot(buttonHit = onButtonHit)
gear = Gear()
robot.addPart(gear)
state = "FORWARD"

while not robot.isEscapeHit():
    if state == "FORWARD":
        robot.drawString("Moving forward", 0, 3)
        gear.forward(moveTime)
        state = "STOPPED"
```

```
        robot.drawString("Teach me!", 0, 3)
    elif state == "LEFT":
        memory.append(0)
        robot.drawString("Saved: LEFT-TURN", 0, 3)
        gear.left(turnTime)
        state = "FORWARD"
    elif state == "RIGHT":
        memory.append(1)
        robot.drawString("Saved: RIGHT-TURN", 0, 3)
        gear.right(turnTime)
        state = "FORWARD"
    elif state == "RUN":
        robot.drawString("Executing memory", 0, 1)
        robot.drawString(str(memory), 0, 2)
        robot.reset()
        robot.drawString("Moving forward", 0, 3)
        gear.forward(moveTime)
        for k in memory:
            if k == 0:
                robot.drawString("Turning left", 0, 3)
                gear.left(turnTime)
            else:
                robot.drawString("Turning right", 0, 3)
                gear.right(turnTime)
            robot.drawString("Moving forward", 0, 3)
            gear.forward(moveTime)
        gear.stop()
        robot.drawString("All done", 0, 3)
        state = "STOPPED"
robot.exit()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Processing the states occurs in a while loop in the main part of the program, instead of in callback functions. In computer science, such a loop is commonly called an **event loop**. Only the corresponding state is selected in the callback (state switch). With this programming technique, you can obtain a clear chronological synchronization between the longer lasting actions and the call of the event-driven callback, which can occur at any given time. The "memory" consists of a list where you store the numbers 0 or 1, depending on whether the path branches off to the left or the right.

THE SELF-LEARNING ROBOT

In certain applications it is not possible for the robot to be taught by an operator. The robot could, for example, be located somewhere outside of the immediate communication area (e.g. on Mars).

In order to find the path, the robot now has to capture the environment using built-in sensors and "act" accordingly. People know the environment mainly from seeing with their eyes. For robots, image capture with a camera is easy, but the analysis of these images can get extremely complicated [**more...**].

In order to orient itself in the canal, your robot uses only a touch sensor that will trigger an event when pressed. The canal should always consist of canal elements of equal lengths. When the robot receives a touch event after passing through a canal member, it knows that it is at a turning point in the path. It then goes back a bit and tries to progress with a left turn.
If within a short amount of time it bumps into a wall again, it knows that it took the wrong path. It then goes back again and this time moves to the right. The robot remembers whether it had to turn to the right or to the left in order to go the right way, and it can later run through the canal

any number of times on its own without bumping into a wall.

Let the robot run through the canal in teach mode. Then, you press the enter key or the ENTER button in order to transfer it from teach mode into execution mode.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

import time

RobotContext.useObstacle("sprites/bg.gif", 250, 250)
RobotContext.setStartPosition(310, 470)
RobotContext.showStatusBar(30)

def onPressed(port):
    global startTime
    global backTime
    robot.drawString("Press event!", 0, 1)
    dt = time.clock() - startTime # time since last hit in s
    gear.backward(backTime)
    if dt > 2:
        memory.append(0)
        gear.left(turnTime) # turning left
    else:
        memory.pop()
        memory.append(1)
        gear.right(2 * turnTime) # turning right
    robot.drawString("Mem: " + str(memory), 0, 1)
    gear.forward()
    startTime = time.clock()

def run():
    for k in memory:
        robot.drawString("Moving forward", 0, 1)
        gear.forward(moveTime)
        if k == 0:
            robot.drawString("Turning left", 0, 1)
            gear.left(turnTime)
        elif k == 1:
            robot.drawString("Turning right", 0, 1)
            gear.right(turnTime)
    gear.forward(moveTime)
    robot.drawString("All done", 0, 1)
    isExecuting = False

moveTime = 3200
turnTime = 545
backTime = 700
memory = []
robot = LegoRobot()
gear = Gear()

robot.addPart(gear)
ts = TouchSensor(SensorPort.S3, pressed = onPressed)
robot.addPart(ts)
startTime = time.clock()
gear.forward()
robot.drawString("Moving forward", 0, 1)

while not robot.isEscapeHit():
    if robot.isEnterHit():
        robot.reset()
        run()
robot.exit()
```

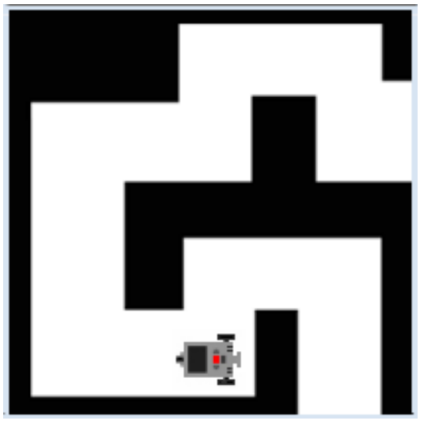
MEMO

The touch sensor is connected to the Port S3 (in the simulation mode, this corresponds to an assembly position in the middle front). The sensor signals the touch events via the callback *onPressed()*, which is registered using the named parameter *press*. The touch sensor is a robot component that is added to the robot as usual with *addPart()*. To find out whether the robot has moved into either a canal element or a dead end, determine the time since the last touch event using the built-in *Python* clock. If it is more than two seconds, the robot moved forward into a canal element, otherwise it was a dead end. Since the robot always turns to the left first and writes a 0 in its brain, in the case of a dead end it has to substitute the incorrect 0 with 1.

THE ENVIRONMENT GETS MORE COMPLEX

You have probably noticed that the robot is perfectly able to figure out by itself how far forward it has to go until the next turn occurs, since it can measure the time that it takes until it bumps into the end of the canal element. This way, the robot is able to move appropriately in a canal with differing lengths of canal elements as well. However, now it must not only keep the left-right information in its head, but also the moving time. You best pack both related pieces of information together in a list *node = [moveTime, k]*, where *moveTime* is the moving time (in ms), *k = 0* is a left turn, and *k = 1* is a right turn.

You will get *moveTime* after passing through the canal element, but you still have to correct it by the time by which the robot has driven too far. Then you store it in a global variable, since you will have to use it again in case the robot moved into a dead end.



```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

import time

RobotContext.useObstacle("sprites/bg2.gif", 250, 250)
RobotContext.setStartPosition(410, 460)
RobotContext.showStatusBar(30)

def pressCallback(port):
    global startTime
    global backTime
    global turnTime
    global moveTime
    dt = time.clock() - startTime # time since last hit in s
    gear.backward(backTime)
    if dt > 2:
        moveTime = int(dt * 1000) - backTime  # save long-track time
        node = [moveTime, 0]
        memory.append(node) # save long-track time
```

```
            gear.left(turnTime) # turning left
        else:
            memory.pop() # discard node
            node = [moveTime, 1]
            memory.append(node)
            gear.right(2 * turnTime) # turning right
        robot.drawString("Memory: " + str(memory), 0, 1)
        gear.forward()
        startTime = time.clock()

def run():
    for node in memory:
        moveTime = node[0]
        k = node[1]
        robot.drawString("Moving forward",0, 1)
        gear.forward(moveTime)
        if k == 0:
            robot.drawString("Turning left",0, 1)
            gear.left(turnTime)
        elif k == 1:
            robot.drawString("Turning right",0, 1)
            gear.right(turnTime)
    gear.forward() # must stop manually
    robot.drawString("All done, press DOWN to stop", 0, 1)
    isExecuting = False

turnTime = 545
backTime = 700

robot = LegoRobot()
gear = Gear()

robot.addPart(gear)
ts = TouchSensor(SensorPort.S3, pressed = pressCallback)
robot.addPart(ts)
startTime = time.clock()
moveTime = 0
memory = []
gear.forward()

while not robot.isEscapeHit():
    if robot.isDownHit():
        gear.stop()
    elif robot.isEnterHit():
        robot.reset()
        run()
robot.exit()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

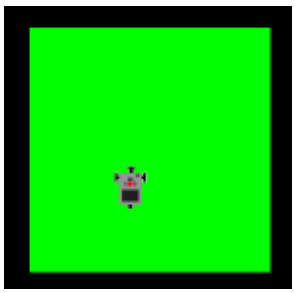
MEMO

At first sight, the data structure of *memory* as a list of *node* lists may seem complicated. However, data belonging together (in this case, the running time of the next track and the subsequent left-right information) should always be stored in a shared data structure. Enjoy that, with the same program, the robot can also find its way in a completely different canal (*bg3.gif*).

EXERCISES

1. The robot should independently pass through a canal shown on the right using a touch sensor, but without learning. For the simulation mode, use the following RobotContext options:

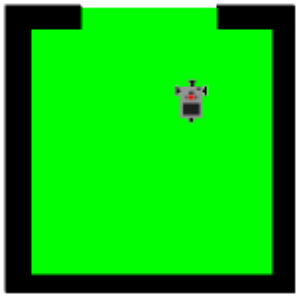
   ```
   RobotContext.useObstacle("sprites/bg2gif", 250, 250)
   RobotContext.setStartPosition(400, 480)
   ```



2. Write a program for a lawn mower robot using a touch sensor that mows the grass strip by strip. It bumps into the boundary of the lawn above and below.

   For the simulation mode, use the following RobotContext options:

   ```
   RobotContext.useBackground("sprites/fieldbg.gif")
   RobotContext.useObstacle("sprites/field1.gif", 250, 250)
   RobotContext.setStartPosition(350, 300)
   ```



3. Unfortunately the boundary of the lawn now has a hole where the robot is able to escape. Create a program with a learning robot that knows how long the strips are after cutting the first strip of grass and thus no longer has to use the touch sensor.

   For the simulation mode, use the following RobotContext options:

   ```
   RobotContext.useBackground("sprites/fieldbg.gif")
   RobotContext.useObstacle("sprites/field2.gif", 250, 250)
   RobotContext.setStartPosition(350, 300)
   ```

ADDITIONAL MATERIAL

TEACH MODE WITH THE INFRARED REMOTE CONTROL

(only EV3 autonomous mode)

You've already learned in the last chapter, how to use the EV3 infrared remote control. Here you uses it to guide the robot in teach mode through the channel. Both in teach as in execute mode, the program runs autonomously on the EV3.

This approach is very close to reality as it is common in industrial robots, they "learn" with a remote control and then execute the "learned" program.

In the teach mode, you use the top two buttons of the remote control to move the robot to the left or right. If you press the lower left button, the execute mode is started. Again, it is elegant to work with states.

```
from ev3robot import *

def onActionPerformed(port, command):
    global state
```

```
    if command == 1:
        state = "LEFT"
    elif command == 3:
        state = "RIGHT"
    elif command == 2:
        state = "RUN"

moveTime = 3200
turnTime = 545
memory = []
robot = LegoRobot()
gear = Gear()
gear.setSpeed(50)
robot.addPart(gear)
irs = IRRemoteSensor(SensorPort.S1, actionPerformed = onActionPerformed)
robot.addPart(irs)
state = "FORWARD"
robot.drawString("Learning...", 0, 3)

while not robot.isEscapeHit():
    if state == "FORWARD":
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "LEFT":
        memory.append(0)
        gear.left(turnTime)
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "RIGHT":
        memory.append(1)
        gear.right(turnTime)
        gear.forward(moveTime)
        state = "STOPPED"
    elif state == "RUN":
        robot.drawString("Executing...", 0, 3)
        robot.reset()
        gear.forward(moveTime)
        for k in memory:
            if k == 0:
                gear.left(turnTime)
            else:
                gear.right(turnTime)
            gear.forward(moveTime)
        gear.stop()
        robot.drawString("All done", 0, 3)
        state = "STOPPED"
robot.exit()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

5.3 CONTROLLING AND REGULATING

INTRODUCTION

When working with machines, which also include robots, we often face the problem of controlling them in a way that a particular measured variable complies as well as possible with a predetermined value (the **target value** or **set point**). For example, the cruise control in a car should keep a predetermined speed even when the car drives on a slope or an incline. To do this, a **control system** with a sensor needs to determine the current speed (the **actual value**) and then adjust the power of the motor accordingly with an actuator, so in a way it needs to operate the gas pedal.

Other examples of technical regulation systems:

- Maintaining the temperature of a refrigerator (thermostat control)
- Keeping an airplane on a specific course (autopilot)
- Maintaining the fill level of a liquid reservoir (e.g. toilet flushing)

Many human activities can be considered as regulatory processes. Some examples:

- Steering a car so that it stays on the road
- Working just as much as you need to barely pass your degree
- Maintaining your balance while standing on one foot

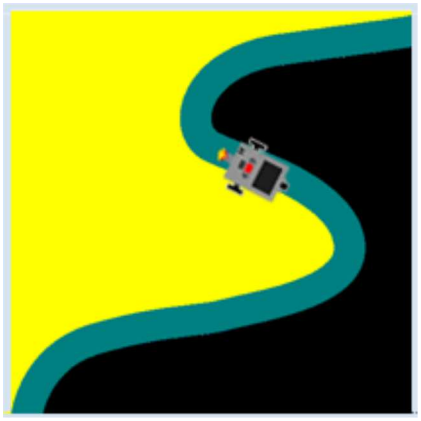
PROGRAMMING CONCEPTS: *Control system, actual value, target value, measurement error*

SELF-DRIVING CAR

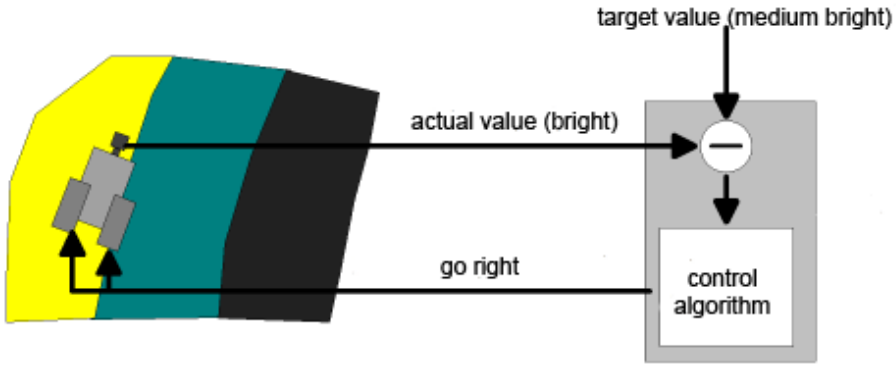
Driving is a complex control process with many input signals that affect the driver not only visually, but also tactually (forces on the body). The mental processing of these signals leads to the driver's behavior (rotation of the steering wheel, pressing the pedals, etc.).

In the future,vehicles will be able to drive themselves without a human, even in complex traffic situations. Several research groups around the world are working on this problem, and it is possible that you might even participate in this interesting research at some point. You can already try out some of your skills here in a highly simplified situation.

Your task is to guide the robot, which is equipped with a chassis and a light sensor that can measure the brightness of the underlying layer, along a green road that is bounded by a yellow and a black area. When on the green part, the sensor signals a middle light value, on the yellow part a large light value, and a small one on the black part. It is the task of the control system to create a control signal for the motors from the measured light values, so that the robot is able to move along the road as well as possible.



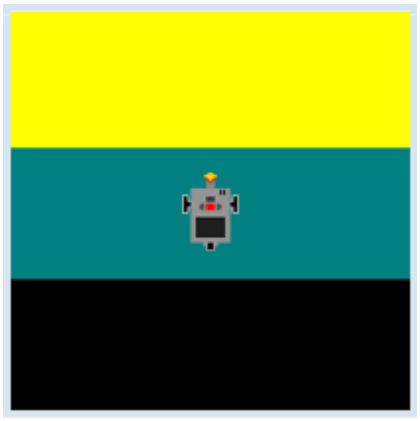
Schematically, you can represent this process in a **control loop**. The light sensor measures the current light value (actual value) and delivers it to the controller, which compares it to the desired value (target value) on the green road. The controller calculates the control quantity for the chassis, meaning that the two motors are switched accordingly, using a control algorithm invented by you, which takes decisions based on the difference between the target and actual values.



Control loop

As you can see, this scheme is indeed a "loop", from the vehicle sensor to the regulatory system, and then back again to the vehicle engines.

Before you can write the program, you need to know the light values that are provided by the sensor for the yellow, green, and black areas. To figure this out, write a small test program that displays the measured values on the console or the display. In the real mode, you do not have to move the robot. Instead, you can just place it on the corresponding area. In the simulation mode, you move through the areas colored accordingly (this process is called **calibration**). Use the NXT Light Sensor for the NXT and the EV3 Color Sensor for the EV3.



```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.setStartPosition(250, 490)
RobotContext.useBackground("sprites/roadtest.gif")

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
```

```
ls = LightSensor(SensorPort.S3)
robot.addPart(ls)
ls.activate(True)
gear.forward()

while not robot.isEscapeHit():
    v = ls.getValue()
    print v
    Tools.delay(100)
robot.exit()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

You are using an obvious control algorithm in your program: If the actual value is larger than the target value, the vehicle is in the yellow area and has to make a right turn. If the actual value is less than the target value, the vehicle is in the black area and has to make a left turn. Otherwise, it can move straight ahead.

```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

RobotContext.setStartPosition(50, 490)
RobotContext.useBackground("sprites/road.gif")

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3)
robot.addPart(ls)
ls.activate(True)
gear.forward()
nominal = 501

while not robot.isEscapeHit():
    actual = ls.getValue()
    if actual == nominal:
        gear.forward()
    elif actual < nominal:
        gear.leftArc(0.1)
    elif actual > nominal:
        gear.rightArc(0.1)

robot.exit()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

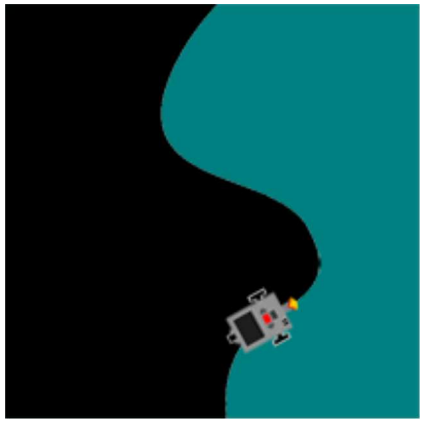
MEMO

The regulation works well in the simulation mode, but not in the real mode. You probably realize why this is: The measured values of the sensor vary, even when the sensor is located on an uniformly colored area. But then this can be expected since the brightness, even on the same surface, never remains exactly the same due to the differences in lighting, and also because of **measurement errors** from the sensor. Try to find a solution for this problem! The curve radius at *leftArc()* or *rightArc()* is a sensitive parameter. A smaller value leads to smaller "outliers" away from the street, but to an unsettled oscillating behavior [**more...**], while a larger value results in a calmer movement, but with less precise guidance on the street. Confirm this through some trials with varied curve radiuses.

EXERCISES

1. Move along a black-green edge with a light sensor. Use the following RobotContext for the simulation mode:

   ```
   RobotContext.useBackground("sprites/edge.gif")
   RobotContext.setStartPosition(250, 490)
   ```



2. Move along a circular path with two light sensors.

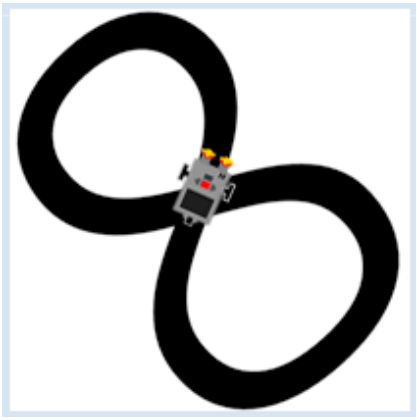
   Use the following RobotContext in the simulation mode:

   ```
   RobotContext.useBackground("sprites/roundpath.gif")
   RobotContext.setStartPosition(250, 250)
   RobotContext.setStartDirection(-90)
   ```

   Change the starting position and direction so that the robot begins on the track.

3. Ride on a roller coaster with two light sensors. Use the background *track.gif* in the simulation mode.

   ```
   RobotContext.useBackground("sprites/track.gif")
   ```



5.4 SENSOR TECHNOLOGY

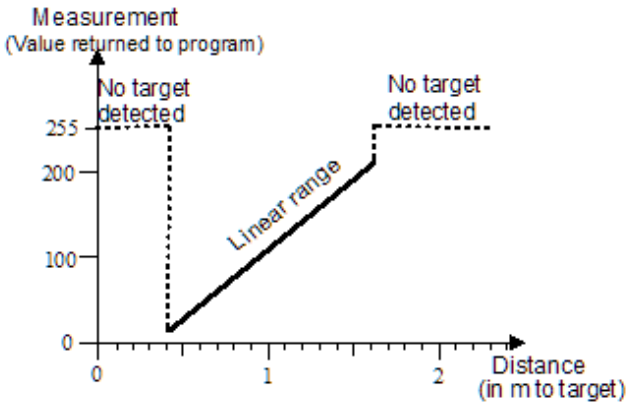
INTRODUCTION

A sensor is a measuring device for a physical quantity such as temperature, light intensity, pressure, or distance. In most cases the value delivered by the sensor can be any number within the measuring range. However, there are also sensors that only know two states, similar to a switch, for example fill level detectors, touch sensors, etc.

The physical quantity in the sensor is usually converted into an electrical voltage and processed further by evaluation electronics [**more...**]. The interior structure of sensors can be highly complex, such as in ultrasonic sensors, gyroscope sensors, or laser distance measurers. The **characteristic curve** of the sensor describes the relationship between the physical measurement value and the value delivered by the sensor. With many sensors the characteristic curve is fairly linear, but one has to determine the conversion factor and the zero offset. For this, the sensor is **calibrated** in a series of measurements with known quantities.

The ultrasonic sensor determines the distance to an object via the running time required for a short ultrasonic pulse to travel from the sensor to the object and back again. For distances between about 30 cm and 2 m, the sensor yields values between 0 and 255, where 255 (in simulation mode -1) is returned when there is no object in the measuring range.

Ultrasonic sensor



Characteristics

In most applications, a sensor is integrated in a program in such a way that its value is periodically retrieved. This is called "**polling the sensor**". In a repeating loop, the sensor values are processed further in the program. The number of measurements per second (temporal resolution) depends on the sensor type, the speed of the computer, and the data connection between the Brick and the program. The ultrasonic sensor is only capable of about 2 measurements per second.

The state of the sensors that have only two states can also be detected through polling. However, it is often easier to conceive of the changing of state as an **event** and to process it programmatically with a **callback**.

PROGRAMMING CONCEPTS: *Sensor, sensor calibration, polling & event, trigger level*

USING POLLING OR EVENTS?

In many cases you can decide whether you would prefer to handle a sensor via polling or events. This is somewhat dependent on the application. You can compare both procedures by connecting a motor and a touch sensor to the brick. Here, a click on the touch sensor should turn the motor

on, and another click should turn it off again.

Events are much smarter for this application because they inform you about the pressing of the touch sensor through a function call. You simply have to pass this function as a named parameter when creating the TouchSensor. With polling, on the other hand, it is necessary to use a flag in order to process only **the transition** from a non-pressed state to a pressed state.

With polling:

```
from nxtrobot import *
#from ev3robot import *

def switchMotorState():
    if motor.isMoving():
        motor.stop()
    else:
        motor.forward()

robot = LegoRobot()
motor = Motor(MotorPort.A)
robot.addPart(motor)
ts = TouchSensor(SensorPort.S3)
robot.addPart(ts)

isOff = True
while not robot.isEscapeHit():
    if ts.isPressed() and isOff:
        isOff = False
        switchMotorState()
    if not ts.isPressed() and not isOff:
        isOff = True
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

With events:

```
#from nxtrobot import *
from ev3robot import *

def onPressed(port):
    if motor.isMoving():
        motor.stop()
    else:
        motor.forward()

robot = LegoRobot()

motor = Motor(MotorPort.A)
robot.addPart(motor)
ts = TouchSensor(SensorPort.S1,
         pressed = onPressed)
robot.addPart(ts)
while not robot.isEscapeHit():
    pass
robot.exit()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Sensors can be handled with polling or events. You must know both of the methods and be able to determine which one is more appropriate in a given situation. In the event model, you define functions whose name usually begins with “on”. These are called **callbacks** because

they are automatically called by the system upon the occurrence of the event ("recalled"). You have to **register** callbacks using named parameters during the generation of the sensor object.

POLLING AN ULTRASONIC SENSOR

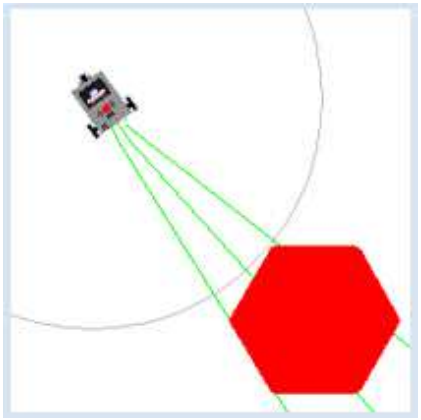
Preliminary note: If you do not have an ultrasonic sensor in your EV3 kit, you can use the EV3 infrared sensor instead.

You must always poll a sensor if you need its measured data at a constant rate. Now you will take on a task where the robot, after you put anywhere on the floor, has to find an object (an aim or target) and travel to it.

You use an ultrasonic sensor to detect a target, which is implemented similarly to a radar target recognition system. To learn about the properties of a sensor and to try it out, you should not shy away from writing a short test program that you will have no need for later on. It is advisable to write out the sensor values and to also make them potentially audible, since you would then have your hands and eyes free to move the robot and the sensor. You request the sensor values in a loop, the period of which adjusts itself, depending on whether you are in autonomous or the external control mode.

```
# from nxtrobot import *
from ev3robot import *

robot = LegoRobot()
us = UltrasonicSensor(SensorPort.S1)
robot.addPart(us)
isAutonomous = robot.isAutonomous()
while not robot.isEscapeHit():
    dist = us.getDistance()
    print "d = ", dist
    robot.drawString("d=" + str(dist), 0, 3)
    robot.playTone(10 * dist + 100, 50)
    if dist == 255:
        robot.playTone(10 * dist + 100, 50)
    if isAutonomous:
       Tools.delay(1000)
    else:
       Tools.delay(200)
robot.exit()
```

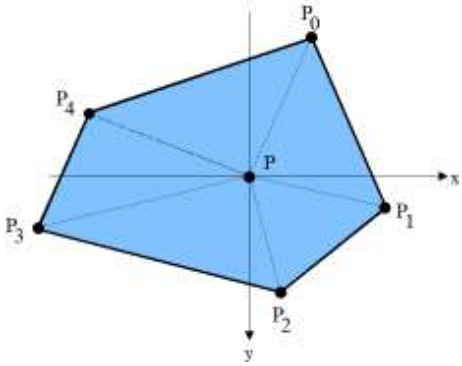
Highlight program code (Ctrl+C to copy, Ctrl+V to paste)



In order to find and get to a target, you rotate the robot like a radar antenna and constantly search for the target with the ultrasonic sensor. If you detect the target, you should note the direction and continue to rotate until the echo stops. You do this in order to determine the apparent size of the target (the angle range in which the target is "visible"). You then move the robot along the middle of the angle of the range and stop at a certain distance.

In simulation mode, you can visualize the distance measuring with **setBeamAreaColor()** and **setProximityCircleColor()**. The displayed target corresponds to the image file that is specified in *RobotContext.useTarget()*.

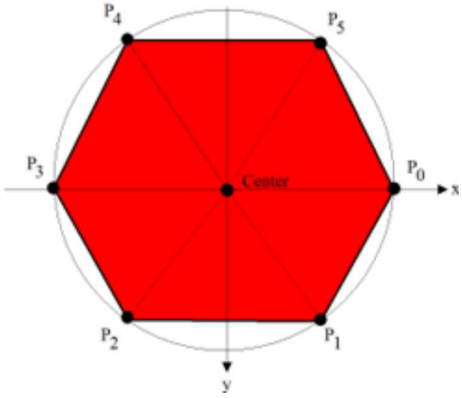
However, for the registration of the target by the simulated sensor it is not that picture that will be used, but a web of triangle meshes. These consists of a common central point and two vertices. The displayed target has the meshes:
PP0P1, PP1P2, PP2P3, PP3P4, PP4P0.



In the program, you indicate the vertices of the meshes as a parameter of the method *useTarget()*. The coordinates refer to a pixel coordinate system with its origin at the center, the positive x-axis pointing to the right, and the positive y-axis pointing downwards.

The mesh coordinates for a hexagon with a diameter of 100 are:

[50, 0] , [25, 43], [-25, 43], [-50, 0], [-25, -43], [25, -43].




```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *

mesh = [[50, 0], [25, 43], [-25, 43], [-50, 0],
          [-25, -43], [25, -43]]
RobotContext.useTarget("sprites/redtarget.gif", mesh, 400, 400)

def searchTarget():
    global left, right
    found = False
    step = 0
    while not robot.isEscapeHit():
        gear.right(50)
        step = step + 1
        dist = us.getDistance()
        print "d = ", dist
        if dist != -1:  # simulation
        #if dist < 80:   # real
            if not found:
                found = True
                left = step
                print "Left at", left
                robot.playTone(880, 500)
        else:
            if found:
                right = step
                print "Right at ", right
                robot.playTone(440, 5000)
                break

left = 0
right = 0
robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
us = UltrasonicSensor(SensorPort.S1)
robot.addPart(us)
us.setBeamAreaColor(makeColor("green"))
us.setProximityCircleColor(makeColor("lightgray"))
gear.setSpeed(5)
```

```
print "Searching..."
searchTarget()

gear.left((right - left) * 25)   # simulation
#gear.left((right - left) * 100)  # real

print "Moving forward..."
gear.forward()

while not robot.isEscapeHit() and gear.isMoving():
    dist = us.getDistance()
    print "d =", dist
    robot.playTone(10 * dist + 100, 100)
    if dist < 40:
        gear.stop()
print "All done"
robot.exit()
```

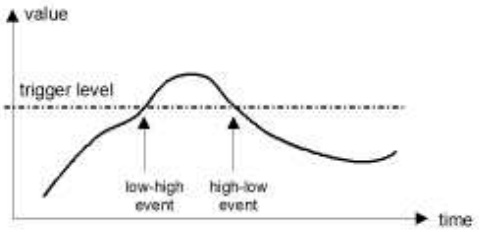
MEMO

You can usually determine the sensor value through the repeated queries (polling) of a getter method (*getValue()*, **getDistance()**, etc.). When switching between simulation mode and real mode you have to adjust certain values, especially time intervals. You must also note that the sensor returns -1 in the simulation mode and 255 in the real mode if it cannot find the target. In simulation mode, the viewing direction of the ultrasonic sensor is determined by the sensor port used:

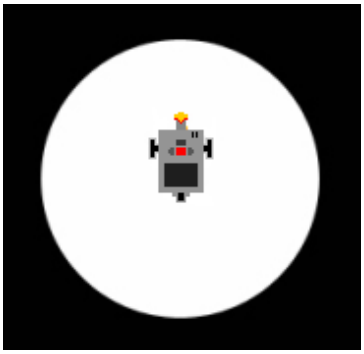
Sensor port	Viewing direction
S1	forwards
S2	left
S3	backwards

EVENTS WITH A TRIGGER LEVEL

Sensors that provide continuous values can be implemented with the event model, too. Here, we define a certain measurement value as a **threshold**, usually called a **trigger level**. An event is triggered when this level is crossed, either from smaller to larger values or vice versa.



The sensors have a default value for the trigger level, but you can change this with *setTriggerLevel()*.Your program ensures that the moving robot stays within a circular area (for example, so that it does not fall off a table). In this case, you use the light sensor, and it must react only to light and dark. If the surface is dark, the callback **onDark** is triggered. With the NXT in real mode, it is important that you turn on the LED illumination of the sensor with **activate(True)**.



```
from simrobot import *
#from nxtrobot import *
#from ev3robot import *
```

```
RobotContext.setStartPosition(250, 200)
RobotContext.setStartDirection(-90)
RobotContext.useBackground("sprites/circle.gif")

def onDark(port, level):
    gear.backward(1500)
    gear.left(545)
    gear.forward()

robot = LegoRobot()
gear = Gear()
robot.addPart(gear)
ls = LightSensor(SensorPort.S3,
      dark = onDark)
robot.addPart(ls)
ls.setTriggerLevel(100)  # adapt value
gear.forward()
while not robot.isEscapeHit():
    pass
robot.exit()
```

MEMO

The crossing of a particular measured value can be interpreted as an event. This is called **triggering**. Default values of the trigger levels:

Sensor	Trigger level(standard)
Sound sensor	50
Light sensor	500
Ultrasonic ensor	10

The advantages and disadvantages of the event model, compared to polling:

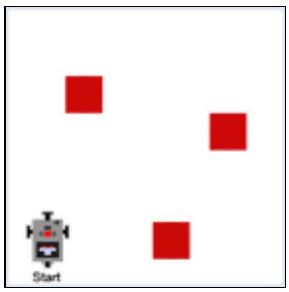
Advantages of the event model	Disadvantages of the event model
A simplified and clearer programming style, since the code in the callback is separate from the rest of the program.	The program currently running is interrupted at unpredictable times (asynchronous). This can interfere with the rest of the program flow.
The event is always detected, even when the PC is slow.	Callbacks can have unwanted side effects, e.g. if they change global variables or the state of the robot.
The main program can continue normally and does not need take care of the sensor.	Callbacks run in a separate process, so there may be conflicts between processes (threads).
Triggering is a central concept of measurement technology.	Only a certain value can be detected (the trigger level).
The event model fits thinking in states (the event puts the system in a new state).	Callbacks should, in principle, only contain short-lasting code, otherwise the other events may get lost.

EXERCISES

1. At a first clap the robot should start moving, and with any further clapping it should change its direction. Solve this problem in both real mode and simulation mode. In real mode, use the sound sensor. In simulation mode, you need a microphone on your PC and you have to correctly set up the microphone level in the control panel.

2. Connect a motor and a touch sensor to the Brick and write a program, where the motor turns on when you press the sensor button and turns off again when you release it.

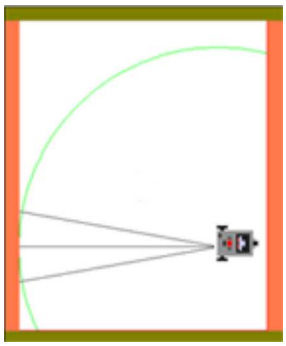
3. Make a robot with an ultrasonic sensor and a touch sensor that finds 3 taller objects (candles, cans...), runs into them, and knocks them over. In simulation mode, you can interpret the knocking over as a touch event and you can use *squaretarget.gif* to represent the objects. The image is 60x60 pixels in size. You can use the following template for the *RobotContext*. Try to understand the information under *mesh*.




```
mesh = [[-30, -30], Point[-30, 30], Point[30, -30], Point[30, 30]]
RobotContext.useTarget("sprites/squaretarget.gif", mesh, 350, 250)
RobotContext.useObstacle("sprites/squaretarget.gif", 350, 250)
RobotContext.useTarget("sprites/squaretarget.gif", mesh, 100, 150)
RobotContext.useObstacle("sprites/squaretarget.gif", 100, 150)
RobotContext.useTarget("sprites/squaretarget.gif" ,mesh, 200, 450)
RobotContext.useObstacle("sprites/squaretarget.gif", 200, 450)
RobotContext.setStartPosition(40, 450)
```

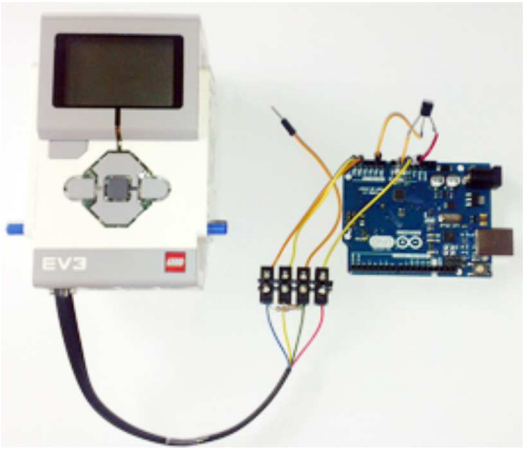
4*. A robot with an ultrasonic sensor is placed at a random position in a rectangular field to begin. Its task is to find the exact middle of the field as quickly as possible. This task can be done in either simulation mode or real mode.

You can use the image file *bar0.gif* and *bar1.gif* as a target.



ADDITIONAL MATERIAL: ARDUINO-SENSORS

In contrast to the EV3 the familiar **Arduino**-microcontroller board has a standard I/O system with digital input and output ports. Analog inputs are also available to connect simple sensors that deliver a voltage proportional to the measured quantity. This allows you to use a variety of inexpensive sensors/actuators and to connect easily home-built electronic circuits. If you connect the EV3 to an Arduino through a suitable communication link, you can access these devices from EV3 programs. The connection between the two systems is simple, if you use an I2C link, since both systems support the I2C protocol.



Here the EV3 acts as I2C master and the Ardunio as I2C slave. The additional software support is already included in the distribution of TigerJython. The EV3 can be operated in direct or autonomous mode. For more information consult the website **http://www.aplu.ch/ev3**.

Documentation robotics

Module import: from simrobot import *
from ev3robot import *
from nxtrobot import *

LegoRobot:

Function	Action
LegoRobot()	generates a robot (without motors) and establishes a connection to the robot
addPart(part)	adds a component to the robot
clearDisplay()	clears the display [*Simulation mode:* status bar]
drawString(text, x, y)	writes text at position x, y [*Simulation mode:* in the status bar, (x, y) is irrelevant]
isEnterHit()	indicates True if ESCAPE button is pressed [*NXT and simulation mode:* use keyboard key *escape*]
isLeftHit()	indicates True if LEFT button is pressed [*NXT and simulation mode:* use keyboard key *cursor left*]
isRightHit()	indicates True if RIGHT button is pressed [*NXT and simulation mode:* use keyboard key *cursor right*]
isUpHit()	indicates True if UP button is pressed [*NXT and simulation mode:* use keyboard key *cursor up*]
playTone(frequency, duration)	plays a tone with a given? frequency (in Hz) and duration (ms) [*Simulation mode:* not available]
setVolume(volume)	sets the volume for all sound output (0..100)
setLED(pattern)	sets the EV3 LEDS: 0: off, 1: green, 2: red, 3: bright red, 4: flashing green, 5: flashing red, 6: flashing bright red, 7: double flashing green, 8: double flashing red, 9: double flashing bright red
exit()	stops the robot and ends the connection
isConnected()	indicates True if the connection is broken, or if the simulation window is closed
reset()	*in simulation mode:* puts the robot to the starting position/direction

Gear:

Gear()	generates a chassis/gear with 2 synchronized motors at port A, B
backward()	moves/drives backwards (non-blocking method)
backward(ms)	moves backwards during a given time (in ms) (blocking method)
isMoving()	indicates True if the chassis is moving
forward()	moves forwards (non-blocking method)
forward(ms)	moves forwards during a given time (in ms) (blocking method)
left()	turns left (non-blocking method)
left(ms)	turns left during a given time (in ms) (blocking method)
leftArc(radius)	moves on a left curve with a given radius (non-blocking method)
leftArc(radius, ms)	moves during a given time (in ms) on a left curve (blocking method)
right()	turns right (non-blocking method)
right(ms)	turns right during a given time (in ms) (blocking method)
rightArc(radius)	moves on a right curve with a given radius (non-blocking method)
rightArc(radius, ms)	moves during a given time (in ms) on a right curve (blocking method)
setSpeed(speed)	sets the speed

stop()	stops the chassis/gears
getLeftMotorCount()	returns current value of left motor counter [not available in sim]
getRightMotorCount()	returns current value of left motor counter [not available in sim]
resetLeftMotorCount()	sets left motor counter to 0 [not available in sim]
resetRightMotorCount()	sets right motor counter to 0 [not available in sim]

TurtleRobot:

TurtleRobot()	generates a robot with a chassis with motors at port A, B
backward()	moves backwards (non-blocking method)
backward(step)	moves the given number of steps backward (blocking method)
forward()	moves forwards (non-blocking method)
forward(step)	moves the given number of steps forward (blocking method)
left()	turns left (non-blocking method)
left(angle)	turns around the given angle to the left (blocking method)
right()	turns right (non-blocking method)
right(angle)	turns around the given angle to the right (blocking method)
setTurtleSpeed(speed)	sets the speed

Motor:

Motor(MotorPort.port)	generates a motor at the motor port A, B, C, or D
backward()	rotates the motor backwards
forward()	rotates the motor forwards
setSpeed(speed)	sets the speed
isMoving()	indicates True if the motor is moving
stop()	stops the motor
getMotorCount()	returns current value of motor counter [not in sim]
resetMotorCount()	sets motor counter to 0 [not in sim]
rotateTo(count)	sets counter to 0, moves motor until count und stops (blocking) [not available in sim]
rotateTo(count, blocking)	same as rotateTo(count), but not blocking for blocking = False [not available in sim]
continueTo(count)	same as rotateTo(count), but counter is not set to 0 [not available in sim]
continueTo(count, blocking)	same as rotateTo(count, blocking), but counter is not set to 0 [not available in sim]
continueRelativeTo(count)	same as continueTo(count), but count is increment [not available in sim]
continueRelativeTo(count, blocking)	same as continueTo(count, blocking), but count is increment [not available in sim]

LightSensor:

LichtSensor(SensorPort.port)	generates a light sensor at the port S1, S2, S3, or S4
LightSensor(SensorPort.port, dark = onDark)	registers the callback function *onDark*
LightSensor(SensorPort.port, bright = onBright)	registers the callback function *onBright*
activate(True)	activates the LED of the light sensor (only when NXT is required)
activate(False)	disconnects the LED of the light sensor
getValue()	indicates the value of the light sensor (a number somewhere between 0 and 1000)
setTriggerLevel(level)	sets a trigger level
bright(port, level), dark(port, level)	callback functions that may be registered by named parameters

ColorSensor:

ColorSensor(SensorPort.port)	generates a color sensor at the port S1, S2, S3, or S4
getColor()	indicates the measured color as a color type with the methods getRed(), getGreen(), and getBlue(), which provides the RGB value from 0 to 255
getColorID()	indicates a color identification number: 1: black, 2: blue, 3:green, 4: yellow, 5: red, 6: white
getColorStr()	provides the color as a string (BLACK, BLUE, GREEN, YELLOW, RED, WHITE and UNDEFINED)
getLightValue()	indicates the brightness (from the HSG model) of the measured color

TouchSensor:

TouchSensor(SensorPort.port)	generates a touch sensor at the port S1, S2, S3, or S4
TouchSensor(SensorPort.port, pressed = onPressed)	registers the callback function *onPressed*
TouchSensor(SensorPort.port, release = onRelease)	registers the callback function *onRelease*
isPressed()	indicates True if the touch sensor is pressed
pressed(port), released(port)	callback functions that may be registered by named parameters

SoundSensor:

SoundSensor(SensorPort.port)	generates a sound sensor at the port S1, S2, S3, or S4
SoundSensor(SensorPort.port, loud = onLoud)	registers the callback function *loudCallback*
SoundSensor(SensorPort.port, quiet = onQuiet)	registers the callback function *quietCallback*
getValue()	indicates the level of the volume (from 0 to 100)
setTriggerLevel(level)	sets a trigger level
loud(port, level), quiet(port, level)	callback functions that may be registered by named parameters

UltrasonicSensor:

UltrasonicSensor(SensorPort.port)	generates a ultrasonic sensor at the port S1, S2, S3, or S4
getDistance()	returns the measured distance (in cm approx.; 255, if measurement fails)
setTriggerLevel(level)	setzt den Triggerlevel (default: 10)
far(port, level), near(port, level)	callback functions that may be registered by named parameters
setProximityCircleColor(color)	Simulation: sets the color of the proximity circle
setMeshTriangleColor(color)	Simulation: sets the color of the mesh triangles
eraseBeamArea()	Simulation: erases the beam area

InfraredSensor (nur EV3):

IRRemoteSensor(SensorPort.port)	creates an infrared sensor for remote control at SensorPort S1, S2, S3 or S4
getCommand(l)	returns the current command ID: 0:Nothing,1: TopLeft,2:BottomLeft,3:TopRight, 4:bottomRight 5:TopLeft&TopRight, 6:TopLeft&BottomRight,7:BottomLeft&TopRight, 8:bottomLeft&BottomRight, 9:Centre,10:BottomLeft&TopLeft,11:TopRight&BottomRight The channel is selected by the red slider switch: 1: top, 4: bottom. It corresponds to the port number, where the sensor is attached.
actionPerformed(port, command)	callback function that may be registered by a named parameter
IRSeekSensor(SensorPort.port)	creates tn infrared search sensor at SensorPort S1, S2, S3 or S4 . The active IR source of the remote control must be switched on (centre button)

v = getValue()	v.bearing returns the direction (-12..12) and v.distance the distance (in cm) to the source. The channel is selected by the red slider switch: 1: top, 4: bottom. It corresponds to the port number, where the sensor is attached
IRDistanceSensor(SensorPort.port)	creates a infrared distance sensorer at SensorPort S1, S2, S3 or S4 (reflecting target)
getDistance()	returns the distance to the target (in cm)

TemperatureSensor (only EV3)):

TemperatureSensor(SensorPort.port)	creates a temperature sensor at SensorPort S1, S2, S3, S4 (Lego NXT Temperature Sensor 9749)
getTemperature()	returns the temperature in range -55..128 degrees Celsius

ArduinoLink (only EV3):

ArduinoLink(SensorPort.port)	creates an I2C master for the connection to the Arduino at SensorPort S1, S2, S3, S4
getReply(request, reply)	sends the request (integer 0..255) to the Arduino and returns the answer in the given list reply (max.16 integers 0..255)
getReplyInt(request)	sends the request (integer 0..255) to the Arduino und returns the answer as integer 0..255
getReplyString(request)	sends the request (integer 0..255) to the Arduino und returns the answer as string (max. 15 ASCII-characters)

RobotContext (nur Simulation)

setStartDirection(angle)	sets the starting direction (0 to east, positive clockwise)
setStartPosition(x, y)	sets the starting position (in pixels, zero at upper-left vertex)
showStatusBar(height)	adds a status bar with given height at the bottom of the window
setStatusText(text)	inserts text into the status bar (old text is erased)
useBackground(filename)	inserts the given image into the background to be used by a light or color sensor
useObstacle(filename, x, y)	inserts an obstacle at given position to be used by a the touch sensor
useTarget(filename, mesh, x, y)	inserts a target at given position to be used by the ultrasonic sensor

chapter six

INTERNET

Learning Objectives

- You know the data type *string* and can work with important string methods.
- You know what a HTML-formatted document is and you also know some HTML tags.
- You can open a HTML document as a file or download it from a web server and display it in a browser window.
- You know the client-server model and can request a file from the web server with the HTTP GET command.
- You know a procedure of how to search a HTML document for specific information.
- You can describe the data type *dictionary* and know in which cases it is especially beneficial.
- You can programmatically perform a search on Google.

6.1 HTML, STRINGS

INTRODUCTION

HTML (Hyper Text Markup Language) is a document description language for websites. A website shown in the browser, however complicated it might appear, is generated from an ordinary text file that contains **markups** for the layout in addition to the visible text. These consist of a **tag** pair with both a start and end tag. The start tag begins with the angle bracket < and closes with the angle bracket >; the end tag starts with </ and is also closed with >.

The basic structure of a HTML text file consists of the tags <*html*> and <*body*> as well as the corresponding end tags.

```
<html>
   <body>
      TigerJython Web-Site
   </body>
</html>
```

The letter case of the tags, as well as the line breaks and indentation, do not matter for the layout of the document.

PROGRAMMING CONCEPTS: *HTML, hyperlink, string, constant data type*

WHAT ARE STRINGS?

In many programs, including in the context of the web, you need a data type in order to store text. This consists in a stringing together of letters (a **character string**) that you can type with the keyboard. In addition, you will need some control characters to do things such as indicating a line break. In Python you use the data type *str* for character strings.

The text of a string is placed between double or single quotes. You can interpret strings as lists whose elements are individual characters. Most familiar operations for lists are also applicable to strings, but with one important difference: You can get a single character from the string with an index (square parentheses), but you cannot change the character with an allocation because the string is a **fixed data type**. If you want to change a string, you have to create a new one.

Your program defines *HTML-formatted* text as a string *html* and writes it out to the console.

```
html = "<html><body>TigerJython Web Site</body></html>"
print html
```

In order to run through a string character by character, you cann use a for loop with an **index**:

```
html = "<html><body>TigerJython Web Site</body></html>"

for i in range(len(html)):
   print html[i]
```

It is more elegant, however, to use a for loop with the keyword **in**:

```
html = "<html><body>TigerJython Web Site</body></html>"
for c in html:
   print c
```

A string can also contain special control characters. These **escape character** are initiated with a **backslash**, for example the character for a new line \n (newline, also called a linefeed <lf>). One example is creating the format shown in the very beginning of the chapter with:

```
html = "<html>\n   <body>\n      TigerJython Web Site\n   </body>\n</html>"
print html
```

You can also read texts from a text file. To do this, create the file *welcome.html* with any text editor in the directory where your program is located in, with the following content:

```
<html>
   <body>
      <h1>TigerJython Web-Site</h1>
      Good morning
   </body>
</html>
```

You draw a heading with the tag <h1>. Your program reads the text file in the *html* string and then writes it out again to the console.

```
html = open("welcome.html").read()
print html
```

MEMO

A string is a constant object consisting of individual characters. You can read individual characters with an index. However, if you try to replace a character with an allocation you will get an error message. There is no character type in Python, since single characters are also considered as strings.

Text files are opened with *open()*. With this, you deliver the path to the file. The path can be relative to the directory where *tigerjython2.jar* is located, but also absolute when you prepend a fraction line (in Windows, possibly also a drive letter), for example:

open("test/welcome.html")	*welcome.txt* in the subdirectory test of the home directory of *tigerjython2.jar*
open("/myweb/test/welcome.html")	*welcome.txt* in the directory */myweb/test* of the drive where *tigerjython2.jar* is located
open("d:/myweb/test/welcome.html")	(only for Windows) *welcome.txt* in the directory *\myweb\test* of the drive d:

You can connect two strings with the addition operator + (concatenate). However, it is important that both operands are really strings. For example, "pi = " + 3.1459 leads to an error message. In this case, you have to write "pi = " + str(3.14159) so that the number is first converted into a string.

The most important operations with strings:

s = "Python"	defines a string (or with the single quotes s = 'Python')
s[i]	accesses string character with index i
s[start:end]	new sub-string of characters start to end, but without end
s[start:]	new sub-string with characters from start
s[:end]	new sub-string with characters from end, but without end
s.index(x)	index of the first occurrence of x (-1: not found)
s.find(x)	index of the first occurrence of x (-1: not found)
s.find(x, start)	index of the first occurrence of x from start
s.find(x, start, end)	index of the first occurrence of x from start to end
s.count(x)	returns the number of occurrences of x
x in s	returns True if x is contained in s
x not in s	returns True if x is not contained in s
s1 + s2	concatenation of s1 and s2 as a new string
s1 += s2	replaces s1 by the concatenation of s1 and s2
s * 4	repeats new string with characters s four times
len(s)	returns the number of characters

WEB BROWSER

The most important task of **web browsers** is to interpret the HTML tags and display the page on a screen window according to the layout information. You can display the file *welcome.html* on your PC after installing a web browser (Firefox, Explorer, Chrome, Safari, Opera, etc.).

TigerJython provides you with a simple browser window as an instance of the class *HtmlPane*. The method **insertText()** causes the input string to appear as a web page in the window.


```
from ch.aplu.util import HtmlPane

html = open("welcome.html").read()
pane = HtmlPane()
pane.insertText(html)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

A web browser interprets the HTML markups and displays the document according to its layout information.

HtmlPane knows only the basic HTML tags. Displaying complex HTML pages is not supported. You can also use a *HtmlPane* to display your program output in a separate window with a pleasing layout, rather than write it out to the console.

HYPERLINKS

The explosive propagation of the web is essentially attributed to the fact that a website can contain elements that lead, by a simple mouse click, to other websites that could be located on any other web server, even far away on the world. Elements of this type are called **hyperlinks**. Hyperlinks can build an interconnected information structure, similar to a spider web.

Create the file *welcomey.html* again with a text editor that contains the link tag <a>. Now we also use the paragraph tag <p> which defines a new section with a line break.

```
<html>
   <body>
      <h1>TigerJython Web-Site</h1>
      <p>Good morning!</p>
      <a href="http://www.tigerjython.ch/">TigerJython Home</a>
   </body></html>
```

You have to enable hyperlinks in your program by defining the function **linkCallback()** (or any other name) and registering it with the named parameter *linkListener*. Clicking on the link leads to the invocation of the callback whereby the URL contained in the link tag is delivered.

```
from ch.aplu.util import HtmlPane

def linkCallback(url):
    pane.insertUrl(url)

html = open("welcomex.html").read()
pane = HtmlPane(linkListener = linkCallback)
pane.insertText(html)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Hyperlinks are cross references in a document with which you can jump to other documents. Linked documents are a characteristic feature of the World Wide Web.

Unfortunately, the display of web pages with *HtmlPane* is incomplete. However, you can use the default browser with *HtmlPane.browse()* [**more...**].

```
from ch.aplu.util import HtmlPane
HtmlPane.browse("www.tigerjython.com")
```

EXERCISES

1. With the tag

```
<img src="gifs/tigerlogo.png" width="120" height="116"></img>
```

you can embed an image that is located in the subdirectory *gifs* of the directory with the HTML file. The values of *width* and *height* should match the image size in pixels.

Create a file *showlogo.html* and a program that shows the following in a *HtmlPane:*

You can download the image *tigerlogo.png* **here**.

2. Define the strings *last name, first name, street,* and *location* as well as the *house number* and *zip code* either with your personal information or with something made up. Link these strings together into a single string *address* with the + sign, so that *print(address)* writes out the formatted information:

first name, last name
house number, street
zip code, location

3. The same information from exercise 2 (above) should appear in a *HtmlPane*, but with the zip code and location in bold.

Write a corresponding HTML-formatted string and deliver it to the *HtmlPane* with *insertText()*.

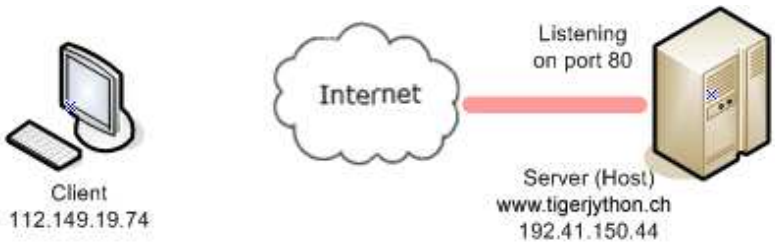
Note: the tag <*br*> creates a line break and the tag <*b*> makes the text bold.

6.2 CLIENT SERVER MODEL, HTTP

INTRODUCTION

You should already know that a web page displayed in your web browser is described by an ordinary HTML text file, which is typically located on an Internet server (also called a **host**). In order to locate the file, the browser uses the URL in the form *http://servername/filepath*.
http stands for Hypertext Transfer Protocol and refers to the process of how the file is transferred from the **server** to your PC browser, called a **client**. The server name, also called its IP address (IP: Internet Protocol), is either in the "dotted" form, e.g. 192.41.150.141, or it is an alias, e.g. www.tigerjython.com. The file path of the HTML file begins with a slash, but it is relative to a specific document path on the server.

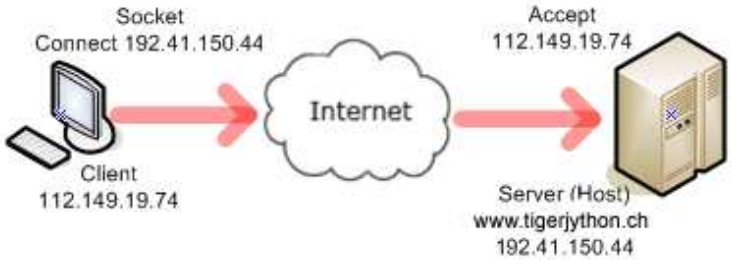
In the communication between the client and the server we employ the **request-response** method, which is one of the most important principles of computer communication. It assumes that a server program is started on the server that waits for a client request on a specific TCP port (for the web port 80).



The exchange of data consists of the following four phases:

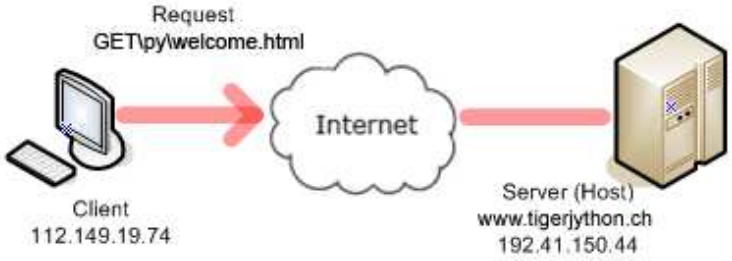
Phase 1:

The client creates a socket object and connects to the server. The server accepts the connection and remembers the client address.



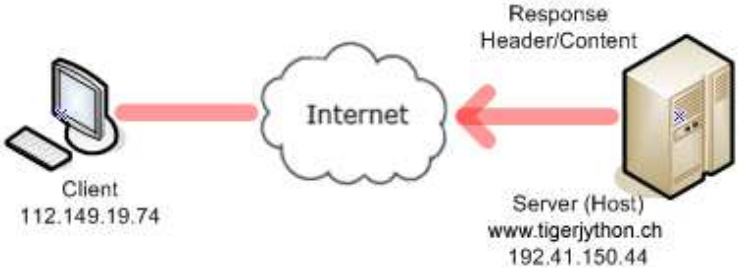
Phase 2:

The client sends a request to the server and passes it the path to the desired file.



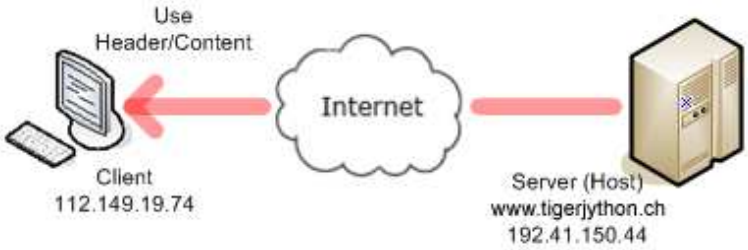
Phase 3:

The server processes the request and sends the file to the client in response.



Phase 4:

The client receives the response and processes it (displayed in the browser window).



PROGRAMMING CONCEPTS: *Host, client, IP address, request-response model, HTTP, parsing*

REQUESTING A WEBSITE WITH HTTP

Your client program performs the phases 1, 2, and 4 and fetches the file *welcomex.html* located in the subdirectory *py* of the server document path.

The method **socket()** of the socket class provides a socket object to the variable *s*. Two constants must be passed that define the correct socket type.

```
import socket
import sys

host = "www.tigerjython.ch"
port = 80
remote_ip = socket.gethostbyname(host)

# Phase 1
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((remote_ip , port))
print "Socket Connected to " + host + " on ip " + remote_ip

# Phase 2
request = "GET /py/welcome.html HTTP/1.1\r\nHost: " + host + "\r\n\r\n"
s.sendall(request)

# Phase 4
reply = s.recv(4096)
print "\nReply:\n"
print reply
```

MEMO

The request for a website from a web server uses the HTTP protocol. This is an agreement between the client and the server and it determines the procedure of the data transfer in full detail. The GET command is documented in the HTTP as follows :

1. Zeile	GET /py/welcomeex.html HTTP/1.1\r\n	The command to fetch the file in the directory path of the server, version of the protocol <carriage return><line feed> (line break)
2. Zeile	Host: hostname\r\n	Name of the host <carriage return><line feed>
3. Zeile	\r\n	Blank line as a marker for the end of the command

HTTP HEADER AND CONNECT

The response consists of a header with status information and the content with the requested file. In order to represent the website, you cut off the header and deliver the contents to a *HtmlPane*.

```
import socket
import sys
from ch.aplu.util import HtmlPane

host = "www.tigerjython.ch"
port = 80
remote_ip = socket.gethostbyname(host)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((remote_ip , port))
request = "GET /py/welcome.html HTTP/1.1\r\nHost: " + host + "\r\n\r\n"
s.sendall(request)
reply = s.recv(4096)

index = reply.find("<html")
html = reply[index:]

pane = HtmlPane()
pane.insertText(html)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The function recv(4096) returns maximum 4096 characters from a data buffer, in which the received characters are copied. To cut away the header use the string method **find(str)**, which searches the string for the given substring *str* and returns the index of the first occurrence. (If the substring is not found, -1 is returned.) Afterwards you can neatly filter out the substring with a slice operation, starting at the index and going to the end.

READING THE WEATHER FORECASTS

You will probably wonder why you should apply such a complicated procedure to display a web page, when you can do the same thing using a single line *insertUrl()* from *HtmlPane*. What you have just learned makes perfect sense, however, if you do not want to display the content of the website in a browser window, for example, but if you are rather interested in only certain information embedded there. As a sensible application your program fetches the current weather forecast from the website of Australian's Bureau of Meteorology and writes it out as text.

You can make your life as a programmer even easier, if you use the library *urllib2* instead of the socket class to fetch the file from the web server [**more...**].

To find out where the desired information is located, you can create an analysis program that simultaneously represents the page as a text in the console window and as a web page in your default browser.

```
import urllib2
from ch.aplu.util import HtmlPane

url = "http://www.bom.gov.au/nsw/forecasts/sydney.shtml"
HtmlPane.browse(url)
html = urllib2.urlopen(url).read()
print html
```

MEMO

The use of software libraries such as *urllib2* simplifies the code, but obscures the basic mechanisms.

PARSING OF TEXTS

You now have the interesting and challenging task of fetching the relevant information from a long string of text, which is the **parsing** of a text.

At first you have the subtask of removing all HTML tags from the string with the function remove_html_tags()

The procedure is typical and the applied algorithm can be described as follows:

You go through the text character by character. You thereby imagine that you remember two **states**: you are either inside or outside of a HTML tag. You should only copy the character to the end of an output string if you are outside of a HTML tag. The change of state takes place while reading the tag angle brackets < or >.

To extract the information of interest you analyze the text after removing the HTML tags by copying it into a text editor and looking for a token that is unique for the beginning of the information. With the string method *find()*, you can get the index start if this token. You then look for a token that characterizes the end of the information and search its index *end* beginning at the *start* index. For this website the start token is "Sydney area" and the end token is "Summary". The text in between is extracted by a slice operation [start:end].

```
import urllib2

def remove_html_tags(s):
    inTag = False
    out = ""

    for c in s:
        if c == '<':
            inTag = True
        elif c == '>':
          inTag = False
        elif not inTag:
            out = out + c
    return out

url = "http://www.bom.gov.au/nsw/forecasts/sydney.shtml"
html = urllib2.urlopen(url).read()
html = remove_html_tags(html)

start = html.find("Sydney area")
end = html.find("Summary", start)
html = html[start:end].strip()

print html

from soundsystem import *

initTTS()
selectVoice("english-man")
sound = generateVoice(html)
openSoundPlayer(sound)
play()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The parsing of texts is usually done character by character. In many cases, however, methods of the string class may help as well [**more...**].

VOICE/SPEECH SYNTHESIS OF WEATHER FORECASTS

With your knowledge from the previous chapter *Sound*, you can have the text of the weather forecast read by a synthetic voice with just a few extra lines of code. You can simply add the following lines to the program:

```
from soundsystem import *

initTTS()
selectVoice("german-man")
sound = generateVoice(html)
openSoundPlayer(sound)
play()
```

EXERCISES

1. Various interesting information is offered under the URL *http://www.timeanddate.com* that can be extracted and reused with a self-written program. One example is that you can get the weather information of any city in the world. Look, for example, at the website *http://www.timeanddate.com/weather/canada/halifax* with a web browser.
   The program should write out the current temperature in an arbitrarily chosen city.

   The procedure: First write out the entire text that you get from the above URL and search for the temperature value. Your program must extract this value using appropriate string methods. Let the user select the country aud town with *inputString()* or an *EntryDialog* and write the temperature value to the console or to a StringEntry of the *EntryDialog*.

2*. The method *urllib2.urlopen(url)* throws an exception in case the URL is not found. Putting the call in a try-except block makes the program branch at an error in the except block.

```
try:
   urllib2.urlopen(url)
except:
   print "Error"
```

   If the city is not found, add a sensible error output to the program from exercise 1.

6.3 BING SEARCH, DICTIONARY

INTRODUCTION

It is possible to use known search engines such as Google, Bing or Yahoo to programmatically perform a web search. For this you have to provide additional parameters to the specific URL of the provider that contain the search string, and perform an HTTP GET request with this. This data is evaluated by a **web application**, i.e. a program that runs on the web server, and the results are returned to you as a HTTP response [**more...**].

Moreover some Web search providers make available search services that can be used via a programming interface (API, Application Programming Interface). Although these services are mostly with costs, there is sometimes a limited, but free version for training and development purposes. For example, using the Bing Search API, you can create your own search machine with an individualized layout.

Search APIs mostly return results in a special format, namely the JavaScript Object Notation (JSON). With the Python module *json* it is easy to convert the format into a Python dictionary. But to extract data of your interest, you have first to learn what is a Python dictionary.

PROGRAMMING CONCEPTS: *Web application, Python dictionary*

UNDERSTANDING A DICTIONARY

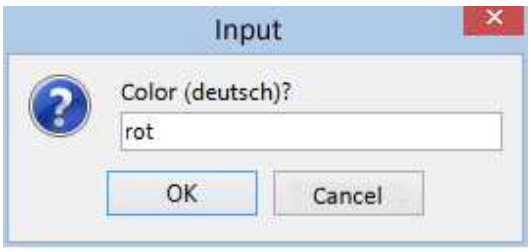
As the name suggests, a dictionary is a data structure similar to a dictionary book. You can imagine word pairs with words on the left being in a language you already know and ones on the right in a foreign language (we disregard any ambiguities). The example below shows some names of colors from English to German:

Deutsch	**Englisch**
blau	blue
rot	red
grün	green
gelb	yellow

(In a real-world dictionary words are arranged alphabetically so that finding a specific word is simplified.)

The word on the left is the **key** and the word on the right is the **value**. A dictionary thus consists of **key-value pairs** [**more..**]. Both keys and values can have any data type [**more...**].

Your program translates the above colors from German to English. If the input is not in the dictionary, the error is caught and an error message appears.



```
dict = {"blau":"blue", "rot":"red", "grün":"green", "gelb":"yellow"}

print "All entries:"
for key in dict:
    print key + " -> " + dict[key]

while True:
    farbe = input("color (deutsch)?")
    if farbe in dict:
        print farbe + " -> " + dict[farbe]
    else:
        print farbe + " -> " + "(not translatable)"
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

A dictionary consists of key-value pairs. In contrast to a list, these pairs are not ordered. In the definition, you use a pair of curly brackets, separate the respective pairs with commas, and key and value with a colon.

Important operations:

dict[key]	provides the value for the key
dict[key] = value	adds a new key-value pair
len(dict)	provides the number of key-value pairs
del dict(key)	deletes the pair (key and value) with the key
key in dict	returns True when the key exists
dist.clear()	deletes all entries, what remains is an empty dictionary

A dictionary can be iterated through with a for loop

```
for key in dict:
```

.

DICTIONARIES ARE EFFICIENT DATA STRUCTURES

You are right if you object to the thought that paired information can be saved in a list. It would be obvious to save each pair as a short list, all of which would be elements of a parent list. Why then is there a dictionary as a separate data structure?
The big advantage of dictionaries is that you can easily and quickly access its values when specifying the key with the bracket notation. So in other words, dictionaries are able to be browsed efficiently. Of course, the efficient retrieval of information only really matters when there are large amounts of data involved, for example when dealing with around a hundred or even thousands of key-value pairs.

As an interesting and useful application, your program should find the postal code of any city in Switzerland. For this, use the text file ***chplz.txt***, which you can download by clicking on the hyperlink. Copy it into the directory where your program is located. The file is structured line by line as follows (and has no blank line, not even at the end):

Aarau:5000
Aarburg:4663
Aarwangen:4912
Aathal Seegraeben:8607
...

Your first task is to convert this text file into a dictionary. In order to do this, first load it in as a string with **read()** and then split it into individual lines using split("\n") [**more...**].

To create the dictionary, separate the key and value in each row once again at the colon and add the new pairs to the (originally empty) dictionary using the bracket notation. Just like before with the colors example, you can now access the postal codes using the bracket notation.

```
file = open("chplz.txt")
plzStr = file.read()
file.close()

pairs = plzStr.split("\n")
print str(len(pairs)) + " pairs loaded"
plz = {}

for pair in pairs:
    element = pair.split(":")
    plz[element[0]] = element[1]

while True:
    town = input("City?")
    if town in plz:
        print "The postal code of " + town + " is " + str(plz[town])
    else:
        print "The city " + town + "was not found."
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

It is very easy and quick to access a value for a certain key in a dictionary [**more...**].

USING BING FOR YOUR OWN PURPOSES

Your program uses the Bing search engine to search for websites with a search string entered by the user and to write out the information provided. In order to access the Bing search machine, you need a personal authentication key. To acquire it, proceed as follows:

Visit the site *https://www.microsoft.com/cognitive-services/en-us/apis* and choose *"Get started for free."* You will be prompted to use your existing Microsoft account or create a new one. In the page titled *Microsoft Cognitive Services* you choose *"APIs"* and *"Bing Web Search"* and click on *"Request new trials"*. Scroll down and select *"Search Bing-Free"*. After confirmation with *"Subscribe"* you get two key values. Save one of them with *copy&paste* for further use a a local text file. You can retrieve the keys any time under your Microsoft account.

In your program you send a GET request supplemented with the search string. The response from Bing is a string in which information is structured by curly brackets. The formatting is consistent with the JavaScript Object Notation (JSON). Using the method **json.load()** it can be converted into a nested *Python* dictionary, that can then be parsed more efficiently. During a test phase, you can analyze the nesting by writing out the appropriate information to the console. You can comment out or remove these lines later. What does Bing find for the search string "Hillary Clinton"?

```
import urllib2
import json

def bing_search(query):
    key = 'xxxxxxxxxxxxxxxxxxxx' # use your personal key
    url = 'https://api.cognitive.microsoft.com/bing/v5.0/search?q=' + query
    urlrequest = urllib2.Request(url)
    urlrequest.add_header('Ocp-Apim-Subscription-Key', key)
    responseStr = urllib2.urlopen(urlrequest)
    response = json.load(responseStr)
    return response

query = input("Enter a search string(AND-connect with +):")
results = bing_search(query)
#print "results:\n" + str(results)
webPages = results['webPages']
print "Number of hits:", webPages["totalEstimatedMatches"]
print "Found URLs:"
values = webPages.get('value')
for item in values:
    print item["displayUrl"]
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

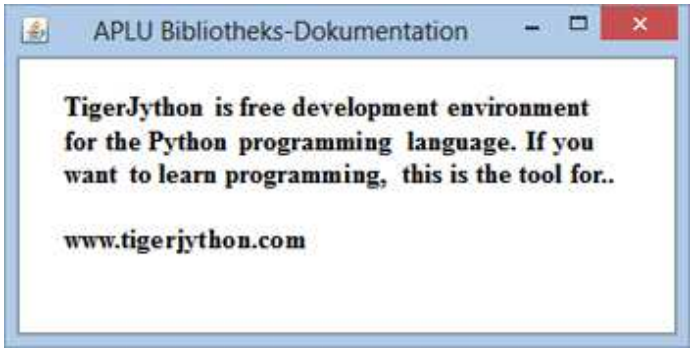
As you can see, a dictionary can in turn contain other dictionaries as values. Thus, hierarchical information structures can be created, similar to XML.

The authentication key is used in a additional header entry of your GET request. You can modify the Bing search by additional query parameters. For example if you append "&count=20" to the URL, you get a total of 20 replies. For more information consult the **API reference**.

EXERCISES

1. Improve the postal code program step by step so that a city will be found even if you:
   a. input spaces before or after the name of the city
   b. do not consistently adhere to the use of upper and lower case letters
   c. write umlauts as Ae, Oe, ae, oe, and ue
   d. omit accents (note: there is a conflict with ö)
   Some places are ambiguous, but have additional information. How will you deal with this?

2. Use Bing Search to write out the title and the content of the search results with the highest ranking in a *HtmlPane*. For example, something similar to the following should appear for the search string "tigerjython":



chapter seven

GAMES & OOP

Learning Objectives

- You know how to define and use a class consisting of a constructor, instance variables, and methods in *Python*.
- You know what a class hierarchy is and how to define and use derived classes.
- You can explain what a polymorphism is in simple words.
- You know the basic design of the game library *JGameGrid* and you can program a simple computer game using it.

"Today, digital media are not imaginable without computer games. Due to the games' high importance for adolescents, many educators are motivated to investigate the didactic potential of the medium. Game-Based Learning (GBL) is the subject of many scientific studies and is part of todays curriculum. In computer science courses, games can be approached from the perspective of the producers. As highly dynamic programs, they encourage beginners to think in procedures and with their moving graphics objects (sprites), they are exceptionally well suited for the introduction into object-oriented programming."

Jarka Arnold, Aegidius Plüss
in "Games as an Introduction to Object-Oriented Programming"

7. 1 OBJECTS EVERYWHERE

INTRODUCTION

In daily life, you are surrounded by a multitude of different objects. Since software often builds of models of reality, it is straightforward to also introduce objects in computer science. This is called **object-oriented programming (OOP)**. For more than decades, the concept of OOP has proved to be groundbreaking in computer science to the extent that it is applied in virtually all developed software systems today [**more...**]. In the following chapter you will learn the main concepts of OOP so that you can participate in the hype.

You have already gotten to know the turtle as an object. A turtle has **properties** (it has a certain color, is located at a certain position, and has a specific viewing direction) and **skills** (it can move forward, rotate, etc.). In OOP, objects with similar properties and skills are grouped in **classes**. The turtle objects belong to the *Turtle class*, we also say that they are **instances** of the *Turtle class*. In order to create an object, you must first **define a class** or use a predefined class such as with the turtles.

When programming, the properties are also called **attributes** or **instance variables**, and the skills are also called **operations** or **methods**. These are variables and functions, except for the fact that they belong to a certain class and are thus "encapsulated" in this class. In order to use them outside of the class, you have to prefix a class instance and the dot operator.

PROGRAMMING CONCEPTS: *Class, object (instance), property, ability, attribute/instance variable, method, inheritance, base/super class, constructor*

AN ADAPTIVE GAME WINDOW

It is not possible to create a computer game with reasonable effort without OOP, since the game performers and other objects of the game board are obviously objects. The game board is a rectangular screen window and it is modeled by the **class GameGrid** from the game library **JGameGrid**. *TigerJython* provides you with an instance when calling **makeGameGrid()** and displays the window with **show()**. Here you can customize the look of the game window with parameter values. With `makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)` a square game window is displayed, the size of which is 10 horizontal by 10 vertical cells of the size 60 pixels. You will see red grid lines and a background image **town.jpg**. (The last parameter causes the navigation bar to be disabled, which is not needed in this case.)

```
from gamegrid import *

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
show()
```

MEMO

The methods of the class *GameGrid* are available to you as functions using *makeGameGrid()*. However, you can also generate an instance yourself and call the methods using the dot operator.

```
from gamegrid import *

gg = GameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
gg.show()
```

The game window is constructed of square cells, and as specified above, each cell is 60 pixels in size with 10 horizontal and 10 vertical cells. This means that when the right and bottom gridline is displayed, the window has a size of 601 x 601 pixels. This corresponds to the (minimum) size of the background image.

The last Boolean parameter determines whether a navigation bar appears.

DEFINING A CLASS WITH DERIVATION

When defining a class, you can decide whether your new class is fully independent, or **derived** from a previously existing class. All the properties and skills of the **parent class** (also called **base class** or **superclass**) are available to you in the derived class. Simply said, the derived class (or **subclass**) **inherits** properties and skills.

In the game library *JGameGrid,* game characters are called **actors** and are instances of the predefined *class actor*. If you need your own performer, **define** a class that is **derived from an actor**.

Your class definition should begin with the keyword **class**. This is followed by any selected class name and a pair of parentheses. In the parentheses, you write down the name of the class from which you derived your class. Since you want to derive your character from *Actor*, you provide that class name.

The class definition contains the definition of the **methods** that are defined like regular functions, with the difference that they **mandatorily** must have the parameter **self** as the first parameter. With this parameter you can access other methods and instance variables of your class and its base class.

The list of method definitions usually begins with the definition of a **special method** named __init__ (with two leading and trailing underlines). This is called a **constructor** and it is automatically called when an object of the class is created. In our case, you call the constructor of the base class *Actor* in the constructor of *Alien,* to which you deliver the path to the sprite image.

Next, you define the method **act()** which plays a central role in game animation, as it is called automatically by the game manager in each simulation cycle. This is a particularly smart trick so that you do not have to worry about animation in a looping structure.

You can define what the game character should do in each simulation cycle with **act()**. As a demonstration, you only move it to the next cell with **move()** in this example. Since *move()* is a method of the base class *Actor*, you have to call it with prefixed *self*.

Once you have defined your class *Alien*, you **create** an alien object by calling the class name and assigning a variable to it. Typical of OOP, you can create as many aliens as you would like. As in everyday life, they each have their own **individuality**, therefore they "know" how they should move from their *act()* method.

To add the created aliens to the game board you use **addActor()**, where you have to specify the cell coordinates with *Location()* (the cell with the coordinates (0,0) is at the top left, x increases to the right, y increases going down). To finally start the simulation cycle, call **doRun()**.

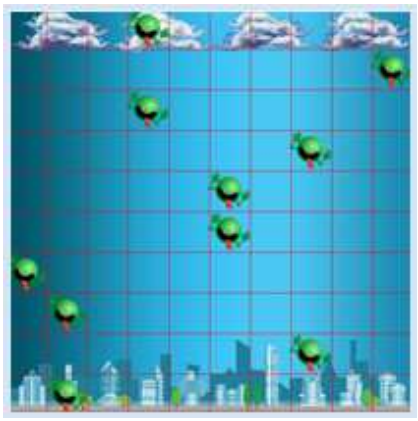
```
from gamegrid import *

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")
    def act(self):
        self.move()
makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
spin = Alien() # object creation, many instances can be created
urix = Alien()
addActor(spin, Location(2, 0), 90)
addActor(urix, Location(5, 0), 90)
show()
doRun()
```

MEMO

A class definition begins with the keyword **class** and **encapsulates** the methods and instance variables of the class. The constructor with the name **__init__** is called automatically when an object is created. To create an object (an instance), write the corresponding class name and pass it the parameter values that __init__ asks for. All game characters are derived from the class *Actor*. You define what the game character should do in each simulation cycle in the method **act()**. With **addActor()** you can add a game character to the game board. You must indicate its starting position (Location) and its starting direction (in degrees) (0 goes east, positive is clockwise).

ATTACK OF THE ALIENS

You notice the strengths and benefits of the object-oriented programming paradigm when you see how easily you are able to populate the game board with many aliens falling from the sky by using just a few lines of code.



Using a repeating loop in the main part, design an alien figure so that every 0.2 seconds a new alien is placed at a random location somewhere in the top grid line.

```
from gamegrid import *
```

```
import random

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False)
show()
doRun()

while not isDisposed():
    alien = Alien()
    addActor(alien, Location(random.randint(0, 9), 0), 90)
    delay(200)
```

MEMO

An endless loop in the main part of the program should test using **isDisposed()** whether the game window was closed, so that the program is able to end correctly.

Note: Sometimes it is necessary to close *TigerJython* and reopen it, so that sprites and background images with the same name but changed content are reloaded.

SPACE INVADERS LIGHT

In your first self-written computer game, the player should try to fight off an alien invasion by removing the invading aliens with a mouse click. Each alien landed in the city subtracts one point.

To get mouse support in the program, you have to add a mouse callback with the name *pressCallback* and register it as the named parameter *mousePressed*. In the callback, you first fetch the cell position of the mouse click from the event parameter *e*. If there is an *Actor* located in that cell you will get it with **getOneActorAt()**. If the cell is empty, None will be returned. **removeActor()** removes the actor from the game board.

```
from gamegrid import *
import random

# ---------------- class Alien ----------------
class Alien(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/alien.png")

    def act(self):
        self.move()

def pressCallback(e):
    location = toLocationInGrid(e.getX(), e.getY())
    actor = getOneActorAt(location)
    if actor != None:
        removeActor(actor)
    refresh()

makeGameGrid(10, 10, 60, Color.red, "sprites/town.jpg", False,
             mousePressed = pressCallback)
setSimulationPeriod(800)
show()
doRun()
```

```
while not isDisposed():
    alien = Alien()
    addActor(alien, Location(random.randint(0, 9), 0), 90)
    delay(1000)
```

MEMO

Since *act()* is called once in every simulation cycle, the period is responsible for the execution speed of the game. The default value of the simulation period is 200 ms. It can be set to a different value using **setSimulationPeriod()**.

The game board is rebuilt in each simulation cycle (rendered), and therefore a change in the game situation is only visible at this moment. If you want to immediately display the new situation at a mouse click, you can manually execute the rendering with **refresh()**.

EXERCISES

1. Create your own background image with an image editor. Add it to the directory sprites, in the same directory where your program is located (or in *<userhome>/gamegrid/sprites*) or specify the fully qualified file path.

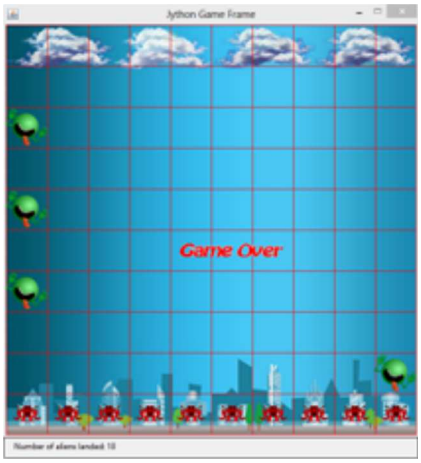
2. Add a 30 pixel high status bar with *addStatusBar(30)* and write the number of aliens that were able to land in the city (despite the defense system) into it using *setStatusText()*.

3. The landed aliens should not simply disappear, rather they should be transformed into a different form at the landing spot ("sprites/alien_1.png" or your own image) and remain there.

   (Hint: with *removeSelf()* you can remove an old alien and with *addActor()* you can generate a new actor in the same place.)

4*. The landed aliens report back to the attacking alien where they have landed so that new aliens can jump into the "open" columns. Once all the columns are occupied, the game ends and displays "Game Over" ("sprites/gameover.gif").

   (Hint: the game manager can be stopped using *doPause()*)



5*. Expand the game with some of your own ideas

7.2 CLASSES AND OBJECTS

INTRODUCTION

.

You have already been acquainted with important concepts of object-oriented programming and noticed that it would be very difficult to write a computer game in Python without OOP. It is therefore important that you get to know the concepts of OOP and their implementation in *Python* a little more systematically.

PROGRAMMING CONCEPTS: *Inheritance, class hierarchy, overriding, is-a relationship, multiple inheritance*

INSTANCE VARIABLES

Animals are well suited to be modeled as objects. First, define a class *Animal* that displays the corresponding animal image in the background of the game board. When creating an object (or an instance) of this class, you pass the file path of the animal image to the constructor so that the method *showMe()* is able to display the image. It does this using the drawing methods of the class *GGBackground*.

The constructor that receives the file path has to save it as an initial value in a variable so that all methods can access it. One such variable is an attribute or an instance variable of the class. In *Python*, instance variables are given the prefix *self* and are generated at the first allocation of a value. As you already know, the constructor has the special name **__init__** (with two leading and trailing underlines). Both the constructor and the methods must have **self** as the first parameter, which is often forgotten.

So, you first define the constructor,

```
def __init__(self, imgPath):
```

as well as a method.

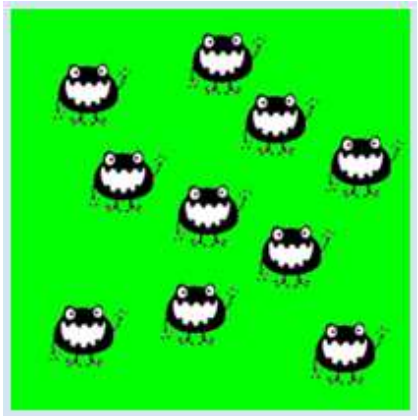
```
def showMe(self, x, y):
```

Once you have generated an animal object *myAnimal* using

```
myAnimal = Animal(bildpfad)
```

you call this method with

```
myAnimal.showMe(x, y)
```



It especially makes sense to use OOP when you are using multiple objects of the same class. To experience this close up, a new animal should pop up in your program at each mouse click.

```
from gamegrid import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath  # Instance variable

    def showMe(self, x, y):  # Method definition
         bg.drawImage(self.imagePath, x, y)

def pressCallback(e):
```

```
    myAnimal = Animal("sprites/animal.gif") # Object creation
    myAnimal.showMe(e.getX(), e.getY())  # Method call

makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
```

MEMO

The properties or attributes of an object are defined as instance variables. They have individual values for each object of the class. Accessing instance variables inside of the class is done by prepending *self*. A class has also access to the variables and functions of the main part of the program, for example all methods of the class GameGrid and with *bg* the backgroundof the game window. The methods can even modify a variable of the main part, if it is declared as global in the method. If the object does not require initialization, the definition of the constructor can also be omitted. Instead of passing the sprite image to the constructor, use the variable *imagePath* in the following program so that you can forego the constructor.

```
from gamegrid import *
import random

# ---------------- class Animal ----------------
class Animal():
    def showMe(self, x, y):
         bg.drawImage(imagePath, x, y)

def pressCallback(e):
    myAnimal = Animal()
    myAnimal.showMe(e.getX(), e.getY())

imagePath = "sprites/animal.gif"
makeGameGrid(600, 600, 1, False, mousePressed = pressCallback)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
```

INHERITANCE, ADDING METHODS

Class hierarchies are created through a class derivation or an inheritance, and with it you can add additional properties and behaviors to an existing class.

Objects of the derived class are also automatically objects of the parent class (also called **base class** or **super class**) and can therefore use all the properties and methods of the parent class as if they were defined in the derived class itself. For example, a pet is an animal that also has its own name, which it is should write out using *tell()*. Hence, you define a class *Pet* that is derived from *Animal*. Since you want to specify the name of the animals for each pet individually during its creation, you provide it to the constructor of *Pet* as an initialization value, which then stores it in an instance variable.


```
from gamegrid import *
from java.awt import Point
```

```
# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath

    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):   # Derived from Animal
    def __init__(self, imgPath, name):
        self.imagePath = imgPath
        self.name = name
    def tell(self, x, y): # Additional method
        bg.drawText(self.name, Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()
bg.setPaintColor(Color.black)

for i in range(5):
    myPet = Pet("sprites/pet.gif", "Trixi")
    myPet.showMe(50 + 100 * i, 100)
    myPet.tell(72 + 100 * i, 145)
```

MEMO

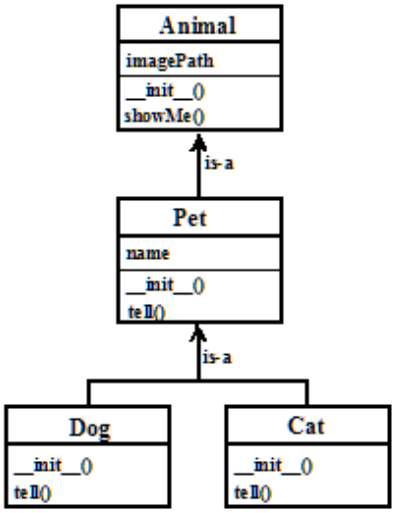
As you can see, you can call *myPet.showMe()* even though *showMe()* is not defined in the class *Pet*, because a pet **is also an animal**. The relationship of *Pet* and *Animal* is therefore called an **is-a relationship**.

Since *imagePath* is set by the *Animal* constructor, you may replace the line *self.imagePath = imgPath* in the Pet constructor by *Animal.__init__(self, imagePath)* to initialize the *Animal* base class.

For derived classes, the base classes are placed in parentheses after the class name. In *Python* you can also derive a class from several base classes (**multiple inheritance**).

CLASS HIERARCHIES, OVERRIDING METHODS

Methods of the base class can also be changed in a derived class by being redefined (overridden) with the same name and parameter list. If you want to model dogs that also bark with *tell(),* derive the class *Dog* from *Pet* and override the method *tell()*. You can get a cat to meow by deriving a class *Cat* from *Pet* and overriding *tell()* there as well.



The four classes can be visualized in a **class diagram**. The is-a relationship becomes particularly clear with it [**more...**].

The classes are displayed as a rectangular box in the class diagram, into which you first write the class name. The instance variables follow separated by a horizontal dividing line and then, led by the constructor, follow the methods of the class. The class hierarchy is easy to follow thanks to a clever arrangement and connecting arrows.

```
from gamegrid import *
from java.awt import Point

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath

    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):
    def __init__(self, imgPath, name):
        self.imagePath = imgPath
        self.name = name
    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- class Dog ----------------
class Dog(Pet):
    def __init__(self, imgPath, name):
        self.imagePath = imgPath
        self.name = name
    def tell(self, x, y): # Overriding
        bg.setPaintColor(Color.blue)
        bg.drawText(self.name + " tells 'Waoh'", Point(x, y))

# ---------------- class Cat ----------------
class Cat(Pet):
    def __init__(self, imgPath, name):
        self.imagePath = imgPath
        self.name = name
    def tell(self, x, y): # Overriding
        bg.setPaintColor(Color.gray)
        bg.drawText(self.name + "  tells 'Meow'", Point(x, y))

makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()

alex = Dog("sprites/dog.gif", "Alex")
alex.showMe(100, 100)
alex.tell(200, 130)  # Overriden method is called

rex = Dog("sprites/dog.gif", "Rex")
rex.showMe(100, 300)
```

```
rex.tell(200, 330)  # Overriden method is called

xara = Cat("sprites/cat.gif", "Xara")
xara.showMe(100, 500)
xara.tell(200, 530)  # Overriden method is called
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

By overriding methods, you can change the behavior of the base class in the derived classes. When calling methods of the same class or the base class, you have to prepend *self*. However, *self* does not have to be provided in the parameter list.

Sometimes you might want to use the identical method of the base class in an override method. To invoke it, you have to prefix the class name of the base class and provide *self* in the parameter list [**more...**].

This rule also applies to the constructor: if the constructor of the base class is used in the constructor of the derived class, it has to be called by prepending the class name of the base class and passing the parameter *self*.

TYPE-BASED METHOD CALLS: POLYMORPHISM

Polymorphism is a bit more difficult to understand, but it is a particularly important feature of object-oriented programming. It refers to the calling of overridden methods, where the call is automatically adjusted to the class affiliation. With a simple example you can see what this means. You use a list *Animals* with the previously defined classes in which there are two dogs and a cat

```
animals = [Dog(), Dog(), Cat()]
```

A problem occurs when going through the list and calling *tell()* because there are three different methods of *tell()* (one in the class *Pet*, *Dog* and *Cat)*.

```
for animal in animals:
    animal.tell()
```

The computer can resolve this ambiguity in one of three ways:

1. It can give an error message.
2. It can call *tell()* of the base class *Pet*.
3. It can find out what kind of pets you have and then call the appropriate *tell()*.

In a polymorphic programming language such as *Python*, the last and best option applies.

```
from gamegrid import *
from soundsystem import *

# ---------------- class Animal ----------------
class Animal():
    def __init__(self, imgPath):
        self.imagePath = imgPath

    def showMe(self, x, y):
         bg.drawImage(self.imagePath, x, y)

# ---------------- class Pet ----------------
class Pet(Animal):
    def __init__(self, imgPath, name):
```

```
        self.imagePath = imgPath
        self.name = name
    def tell(self, x, y):
        bg.drawText(self.name, Point(x, y))

# ---------------- class Dog ----------------
class Dog(Pet):
    def __init__(self, imgPath, name):
         self.imagePath = imgPath
         self.name = name
    def tell(self, x, y): # Overridden
         Pet.tell(self, x, y)
         openSoundPlayer("wav/dog.wav")
         play()

# ---------------- class Cat ----------------
class Cat(Pet):
    def __init__(self, imgPath, name):
        self.imagePath = imgPath
        self.name = name
    def tell(self, x, y): # Overridden
        Pet.tell(self, x, y)
        openSoundPlayer("wav/cat.wav")
        play()


makeGameGrid(600, 600, 1, False)
setBgColor(Color.green)
show()
doRun()
bg = getBg()

animals = [Dog("sprites/dog.gif", "Alex"),
     Dog("sprites/dog.gif", "Rex"),
     Cat("sprites/cat.gif", "Xara")]

y = 100
for animal in animals:
    animal.showMe(100, y)
    animal.tell(200, y + 30)    # Which tell()????
    show()
    y = y + 200
    delay(1000)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

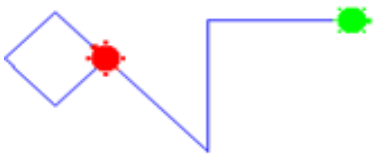
Polymorphism ensures that the class affiliation decides which method is called in overridden methods. Since the affiliation to classes in *Python* is only determined at runtime anyway, polymorphism is self-evident.

The dynamic data binding of Python is called **duck test** or **duck typing**, according to the quote attributed to James Whitcomb Riley (1849 – 1916):
"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

There are some cases where an overridden method is defined in the base class, but it should not do anything. This can be achieved either with an immediate **return** or with the empty statement **pass**.

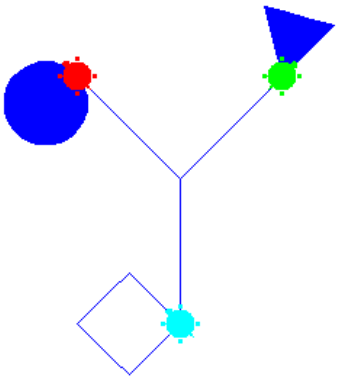
EXERCISES

1. Define a class *TurtleKid* derived from the class *Turtle* that draws a square with *shape()*. Try to get the following main part to work:



```
tf = TurtleFrame()
# john is a Turtle
john = Turtle(tf)
# john knows all commands of Turtle
john.setColor("green")
john.forward(100)
john.right(90)
john.forward(100)

# laura is a TurtleKid, but also a Turtle
# laura knows all commands of Turtle
laura = TurtleKid(tf)
laura.setColor("red")
laura.left(45)
laura.forward(100)
# laura knows a new command too
laura.shape()
```

2. Define two derived classes, *TurtleBoy* and *TurtleGirl,* from *TurtleKid* which override *shape()* so that a *TurtleBoy* draws a solid triangle and a *TurtleGirl* draws a solid circle. The following main part has to work:



```
tf = TurtleFrame()

aGirl = TurtleGirl(tf)
aGirl.setColor("red")
aGirl.left(45)
aGirl.forward(100)
aGirl.shape()

aBoy = TurtleBoy(tf)
aBoy.setColor("green")
aBoy.right(45)
aBoy.forward(100)
aBoy.shape()

aKid = TurtleKid(tf)
aKid.back(100)
aKid.left(45)
aKid.shape()
```

3. Draw the class diagram for exercise 2.

ADDITIONAL MATERIAL

STATIC VARIABLES AND STATIC METHODS

Classes can also be used to group related variables or functions, thus making the code easier to read. For example, you can condense the main physical constants in the class *Physics*. Variables defined in the same class header are called **static variables** and we call them by prefixing the class name. Unlike with instance variables, it is not necessary to create an instance of the class.

```
import math

# ---------------- class Physics ----------------
class Physics():
    # Avogadro constant [mol-1]
    N_AVOGADRO = 6.0221419947e23
```

```
    # Boltzmann constant [J K-1]
    K_BOLTZMANN = 1.380650324e-23
    # Planck constant [J s]
    H_PLANCK = 6.6260687652e-34;
    # Speed of light in vacuo [m s-1]
    C_LIGHT = 2.99792458e8
    # Molar gas constant [K-1 mol-1]
    R_GAS = 8.31447215
    # Faraday constant [C mol-1]
    F_FARADAY = 9.6485341539e4;
    # Absolute zero [Celsius]
    T_ABS = -273.15
    # Charge on the electron [C]
    Q_ELECTRON = -1.60217646263e-19
    # Electrical permittivity of free space [F m-1]
    EPSILON_0 = 8.854187817e-12
    # Magnetic permeability of free space [ 4p10-7 H m-1 (N A-2)]
    MU_0 = math.pi*4.0e-7

c = 1 / math.sqrt(Physics.EPSILON_0 * Physics.MU_0)
print("Speed of light (calulated): %s m/s" %c)
print("Speed of light (table): %s  m/s" %Physics.C_LIGHT)
```

You can also group a collection of related functions by defining them as static methods in a meaningfully designated class. You can use these methods by directly prepending the class name, without having to create an instance of the class [**more...**] .

To make a static method, you have to write the line *@staticmethod* before the definition.

```
# ---------------- class OhmsLaw ----------------
class OhmsLaw():
    @staticmethod
    def U(R, I):
        return R * I
    @staticmethod
    def I(U, R):
        return U / R
    @staticmethod
    def R(U, I):
        return U / I
r = 10
i = 1.5
u = OhmsLaw.U(r, i)
print("Voltage = %s V" %u)
```

MEMO

Static variables (as opposed to instance variables, also called **class variables**) belong to the class as a whole and in contrast to instance variables, all objects of the class have the same value. They can be read and changed with prepended class names.

A typical use of static variables is an **instance counter**, which is a variable that counts the number of generated objects of the relevant class.

Related functions can be grouped as static methods in a suggestively designated class. The line @staticmethod (called a **function decorator**) must be prepended when defining the function.

7.3 ARCADE GAMES, FROGGER

INTRODUCTION

Many computer games found on game consoles and the Internet consist of images moving over a background. Sometimes even the background moves too, especially in situations where the game characters move to the edge of the window allowing the player to have the impression of a scenario that is substantially larger than the display window. Even though game animation requires a lot of computational power, it is generally easy to understand: at time points in quick succession, the so-called game loop recalculates the screen content, copies the background and the images of the game characters into an invisible image buffer, and then displays (renders) it as a whole in the window. With anything more than around 25 frames per second, you will see a smooth, flowing movement, but with anything less the movement will be jerky.

In many games, characters interact through collisions, and therefore dealing with collisions is fundamental. Well developed game libraries such as *JGameGrid* provide programmers with built-in collision detection under the use of event models. One defines the potential collision partners and the system automatically calls a callback when a collision occurs.

PROGRAMMING CONCEPTS: *Game design, sprite, actor, collision, supervisor*

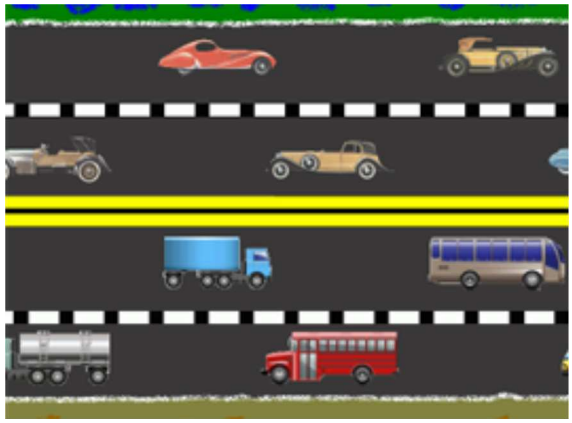
GAME-SCENARIO

When developing a computer game it is important that you first think of a game scenario that is as detailed as possible, and then write out notes as functional program specifications. Often your goals are set too high in your first attempt, and so you should try to simplify the game a bit so that you can develop executable subversions that can then gradually be expanded. The trick is to write the program as generally as possible so that you will not have to modify the existing code with following extensions. Instead, you will only need to add to it. However, this rarely succeeds right off the bat, even with professional programmers, and it is thus common that feelings of euphoria and frustration lie close together when developing a game. This will make your pleasure and satisfaction even greater once you can finally show off your very own personal computer game and let people play it.

Along the way of becoming a competent game programmer, it is advisable to learn by developing well-known games that you can implement in your own personal way using your own sprite images. It is not very important that these games are readily available in the training phase, because it is not about playing a lot, but rather about learning how it is developed. One well-known game is Frogger. It has the following fun scenario:

A frog tries to move across a busy road and get to a pond. If it hits a car, the frog loses its life. The goal is to use the cursor keys to bring it safely across the road.

You should implement four lanes: two of the lanes with trucks and buses moving in opposite directions, and the other two with vintage cars (see the adjacent image).



There are two possible development paths available: you can either first implement the movement of the frog, or the movement of the vehicles. After this, you add the collision mechanism and the calculation of the game points, as well as the end of the game (game over).

In *GameGrid* the vehicles are modeled as instances of the class *Car* derived from *Actor*. The movement of the vehicles is programmed in the method **act()**.

Use the images *car0.gif,..car19.gif* as sprites, located in the distribution of *TigerJython*. You can, of course, also use your own images (they should be a maximum of 70 pixels high and 200 pixels wide with a transparent background).

With arcade games it is common to use a game board with a size of 1 pixel, i.e. the grid corresponds to the pixel grid. Choose 800 x 600 pixels as a window size and display the road scenario using the background image *lane.gif* with the size 801 x 601 pixels. Generate 20 car objects in the function **initCars()** and think about where and with which viewing direction you want to add them to the game board.

Moving the cars with the method **act()** is easy: you push them on with *move(),* and let the cars moving from left to right jump to the far left as soon as they have driven out of the window, and similarly let the cars moving from right to left jump to the right. Remember that the location coordinates of actors can also be outside of the screen.

```
from gamegrid import *

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)


makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False)
setSimulationPeriod(50)
initCars()
show()
doRun()
```

MEMO

Usually a game grid with a cell size of 1 pixel is used in arcade games (pixel game).
The game scene is rendered 20 times per second in a simulation period lasting 50 ms, resulting in a relatively good flow of movement. Sporadic jerks are caused by a computer's low processing power. Due to the limit of computing power, the simulation period cannot be significantly reduced.

MOVING THE FROG WITH THE CURSOR KEYS

Now you can start thinking about how to incorporate the frog into the game. It should first appear at the bottom of the screen, and should then move with the up, down, left, and right cursor keys.

Since the frog is also an *Actor*, first write the class *Frog* which you derive from *Actor*. You do not need any methods besides the constructor, since the frog is moved by keyboard events. For this, define the callback **onKeyRepeated**, that you register through the call *makeGameGrid()* with the parameter named *keyRepeated*. This callback is not only called once when you press the key, but also periodically if you hold it down.

You test the key code in the callback and move the frog 5 steps further accordingly.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
  def __init__(self):
      Actor.__init__(self, "sprites/frog.gif")

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        if i < 5:
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)


makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
             keyRepeated = onKeyRepeated)
setSimulationPeriod(50);
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

MEMO

In order to capture keyboard events, you can also register the callbacks *keyPressed(e)* and *keyReleased(e)*. In contrast to *keyRepeated(code)*, the key code must be fetched from the parameter *e* with *e.getKeyCode()*. Moreover, *keyPressed(e)* is less suitable in this game because there is a delay after pressing and holding down the button until the following press events are triggered.

If you do not know the key codes, you should write a small test program that writes them out:

```
from gamegrid import *

def onKeyPressed(e):
    print "Pressed: ", e.getKeyCode()

def onKeyReleased(e):
    print "Released: ", e.getKeyCode()

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
  keyPressed = onKeyPressed, keyReleased = onKeyReleased)
show()
```

COLLISION EVENTS

The procedure to detect collisions between actors is simple: When generating a vehicle *car*, you say that the frog should trigger an event when colliding with a car, using the method

`frog.addCollisionActor(car)`

The collision event triggers the method **collide()**, located in the class *Frog*. There, you treat the event according to your own wishes, for example you could make the frog jump back to the starting position.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")
        self.setCollisionCircle(Point(0, -10), 5)

    def collide(self, actor1, actor2):
        self.setLocation(Location(400, 560))
        return 0

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
        if i < 5:
```

```
            addActor(car, Location(350 * i, 100), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
     keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()
```

MEMO

The method **collide()** is not an actual callback, but rather a method of the class *Actor* that is overridden in *Frog*. This is why you do not need to register *collide()* with a named parameter.

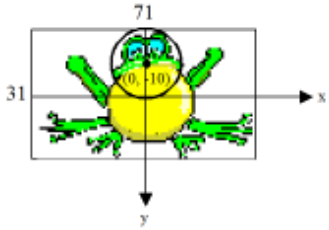
By default, the collision event is triggered when the bounding rectangles of the sprite images overlap. However, you can also change the collision areas in shape, size, and position to fit the sprite image. For this, you can use the following methods of the class *Actor*:

Methode	**Collision area**
setCollisionCircle(centerPoint, radius)	circle with a given center and radius (in pixels)
setCollisionImage()	non-transparent image pixels (only with a partner that has a circle, line, or point as a collision area)
setCollisionLine(startPoint, endPoint)	line between the given start and end points
setCollisionRectangle(center, width, height)	rectangle with a given center, width, and height
setCollisionSpot(spotPoint)	an image pixel

All methods use a relative pixel coordinate system with the zero point in the center of the sprite image, a positive x-axis going to the right, and a positive y-axis going down.

The frog image is 71 x 41 pixels in size. So, for example, you can add the following to the constructor of *Frog*

```
self.setCollisionCircle(Point(0, -10), 5)
```



so that a vehicle has to drive over the circle with a radius of 5 pixels around the head of the frog to trigger a collision event.
(Since the collision area is cached for efficiency reasons, it may be necessary to restart *TigerJython* so that your changes are taken into effect.)

GAME SUPERVISOR AND SOUND

In many games it is necessary that an "independent game supervisor" is made responsible for the compliance with the game rules, the distribution of points, and the proclamation of the winner at game over. Similar to daily life, it is also better if the task is not allocated to a character in the game, but rather to an independent part of the program. The main part of the program is especially well suited for this, which continues to runs after the initialization of the game. Add a loop to the end of the existing program so that it periodically checks the game and reacts accordingly. You should, however, not implement a very tight loop without a **delay()**, as this will unnecessarily waste computing power, which can lead to delays in the remaining execution of the program. The loop should stop when the game window is closed. You can ensure this with **isDisposed = True**. The supervisor can, for example, limit the number of attempts and also count and display the number of successful and unsuccessful crossings of the road.

Dealing with the *game over* situation is often especially tricky, since one has to consider different variants. It is also often the case that you want to play the game several times without restarting the program.

You can use your knowledge from the chapter *Sound* to include sound effects. The easiest way to do this is to use the function *playTone()*.

```
from gamegrid import *

# ---------------- class Frog ----------------
class Frog(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/frog.gif")

    def collide(self, actor1, actor2):
        global nbHit
        nbHit += 1
        playTone([("c''h'a'f'", 100)])
        self.setLocation(Location(400, 560))
        return 0

    def act(self):
        global nbSuccess
        if self.getY() < 15:
            nbSuccess += 1
            playTone([("c'e'g'c''", 200)])
            self.setLocation(Location(400, 560))

# ---------------- class Car ----------------
class Car(Actor):
    def __init__(self, path):
        Actor.__init__(self, path)

    def act(self):
        self.move()
        if self.getX() < -100:
            self.setX(1650)
        if self.getX() > 1650:
            self.setX(-100)

def initCars():
    for i in range(20):
        car = Car("sprites/car" + str(i) + ".gif")
        frog.addCollisionActor(car)
        if i < 5:
            addActor(car, Location(350 * i, 90), 0)
        if i >= 5 and i < 10:
            addActor(car, Location(350 * (i - 5), 220), 180)
        if i >= 10 and i < 15:
            addActor(car, Location(350 * (i - 10), 350), 0)
```

```
        if i >= 15:
            addActor(car, Location(350 * (i - 15), 470), 180)

def onKeyRepeated(keyCode):
    if keyCode == 37: # left
        frog.setX(frog.getX() - 5)
    elif keyCode == 38: # up
        frog.setY(frog.getY() - 5)
    elif keyCode == 39: # right
        frog.setX(frog.getX() + 5)
    elif keyCode == 40: # down
        frog.setY(frog.getY() + 5)

makeGameGrid(800, 600, 1, None, "sprites/lane.gif", False,
    keyRepeated = onKeyRepeated)
setSimulationPeriod(50)
setTitle("Frogger")
frog = Frog()
addActor(frog, Location(400, 560), 90)
initCars()
show()
doRun()

# Game supervision
maxNbLifes = 3
nbHit = 0
nbSuccess = 0
while not isDisposed():
    if nbHit + nbSuccess == maxNbLifes:  # game over
        addActor(Actor("sprites/gameover.gif"), Location(400, 285))
        removeActor(frog)
        doPause()
    setTitle("#Success: " + str(nbSuccess) + " #Hits " + str(nbHit))
    delay(100)
```

MEMO

The counting of successes with **nbSuccess** and failures with **nbHit** takes place in the class *Frog*. This is why these variables have to be declared as globals. You could also use static or instance variables of the *Frog* class. At game over an *Actor* image with a text is inserted, the frog is removed, the simulation cycle is stopped with **doPause()**, and finally, the supervisor loop is left with break. You could also use a *TextActor,* which makes it possible to adjust the text at runtime.

```
rate = nbSuccess / (nbSuccess + nbHit)
ta = TextActor("  Game Over: Success Rate = " + str(rate) + " % ",
     DARKGRAY, YELLOW, Font("Arial", Font.BOLD, 24))
addActor(ta, Location(200, 287))
```

EXERCISES

1. Replace the background image and the vintage car photos with animal images that swim in a river (crocodiles, etc.).

2. Introduce a scoring system and a time limit for the crossing: Each successful crossing should give you 5 points, each hit should take away 5 points. Exceeding the time limit should minus 10 points and put the frog back at the starting point.

3. Add some of your own ideas to the game.

7.4 GRID GAMES, SOLITAIRE BOARD GAME

INTRODUCTION

In a certain class of computer games, tokens are restricted to be located on cells in a grid structure whereby the cells often have the same size and are arranged in a matrix. The consideration of this location restriction on a grid structure substantially simplifies the implementation of the game. As the name implies, the game library *JGameGrid* is particularly optimized for grid-like games.

In this chapter you will gradually develop the peg solitaire with the English board layout. You will get to know important solution methods that you can apply to all grid games.

PROGRAMMING CONCEPTS: *Game board, game rules, specifications, game over*

BOARD INITIALIZATION, MOUSE CONTROL

There is a regular arrangement of holes or recesses in a board into which you can either plug pegs or put marbles. The best-known Solitaire board uses a board with a cross-like arrangement of 33 holes and is called the English board. At the start of the game, all the holes except for the center hole are filled with marbles. As the name *Solitaire* implies, the game is usually played by a single person.

An English *Solitaire* board from India, 1830
© 2003 puzzlemuseum.com

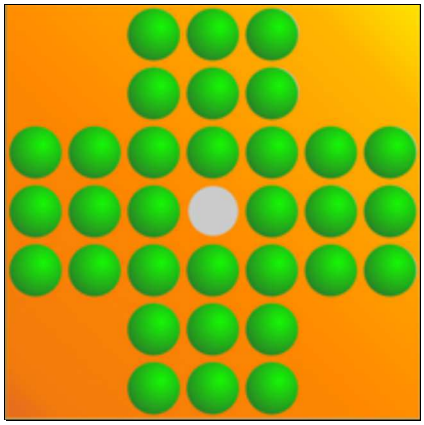
The following rules apply: a turn consists of moving a marble onto a free hole by skipping exactly one marble either horizontally or vertically. The skipped marble is removed from the game board.

The goal is to "clear up" all the marbles from the board, except for the last marble. If the last marble ends up in the center, the game is considered to be solved especially well. When Solitaire is implemented as a computer game, you should be able to "grab" a certain marble by pressing the mouse button and move it by holding down and dragging. When you release the mouse button, the game checks if the turn followed the rules of the game. If you make an illegal move the marble will jump back to its previous location, and if you make a legal move the marble will appear at the new location and the skipped marble will be removed from the board.

With this, the specification is clear and you can start with the implementation. As always, this is done step by step, and you should make sure that your program is running at each of these steps. It is perfectly obvious to use a game grid with 7x7 cells, without using the corner cells. First you draw the board in the function *initBoard()* using the background image *solitaire_board.png* which is included in the distribution of *TigerJython*.You implement the mouse controls with the mouse callbacks *mousePressed*, *mouseDragged*, and *mouseReleased*.

At a **Press event** you keep track of the current cell location and the marble currently in it. You can obtain the marble with **getOneActorAt()** and you will receive *None* if the cell is empty. If you write out important results in a status bar (or in the console), the development process will be easier to control and mistakes easier to find.

During the **Drag event** you move the visible image of the marble to the current cursor position using **setLocationOffset()**. You also move it to any mouse position away from the middle of the cells, so that a continuous motion arises. It is important that this does not move the marble actor itself, but only its sprite image (hence the term *offset*). With this, you can avoid any difficulties with superimposed actors.



In this first version, the marble should simply jump back to its original position upon a **Release event**. You can do this by calling **setLocationOffset(0, 0).**

```
from gamegrid import *

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if loc.x in [0, 1, 5, 6] and loc.y in [0, 1, 5, 6]:
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ". No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ". Marble found")

def dragEvent(e):
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if movingMarble == None:
        return
    movingMarble.setLocationOffset(0, 0)

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
    mousePressed = pressEvent, mouseDragged = dragEvent,
    mouseReleased = releaseEvent)
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

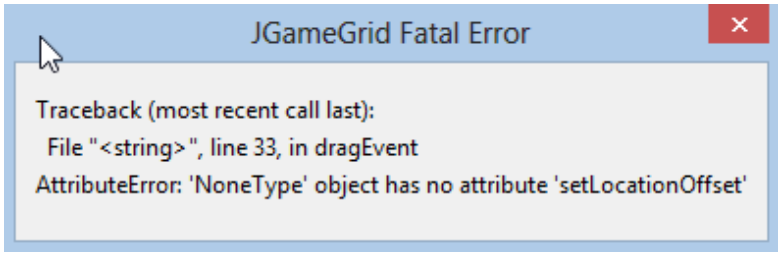
Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Instead of moving the actual actor while dragging, you can just move its sprite image. For this, use **setLocationOffset(x, y)** where x and y are coordinates relative to the current center point of the actor.

You have to carefully distinguish between the coordinates of the mouse and the cell coordinates when dealing with mouse movements. You can use the functions *toLocationInGrid(pixel_coord)* and *toPoint(location_coord)* to convert between these coordinates.

If you start at an empty cell, the *drag* and *release events* lead to an infamous program crash. This is because you are trying to call a method with *movingMarble* that has the value *None*.



In order to avoid this error, leave the callbacks with an immediate return right at the beginning.

IMPLEMENTING THE GAME RULES

How would you verify the game rules with the real game? You would have to know which marble you started with, so you would need to know its starting location *start*. You would then need to know where you want to move the marble to, which is the cell location *dest*. The following conditions must be met in order to make a legal move:

1. At *start* there is a marble
2. At *dest* there is no marble
3. *dest* is a cell belonging to the board
4. *start* and *dest* are either horizontally or vertically two cells apart
5. There is a marble in the cell between them, too

It is a good idea to implement these conditions in a function **getRemoveMarble(start, dest)** that returns the marble to be removed after a legal turn and returns *None* after an illegal turn.

Thus, you should call this function at the release event and remove the returned *Actor* from the board using **removeActor()** if the turn was legal.

```
from gamegrid import *

def getRemoveMarble(start, dest):
    if getOneActorAt(start) == None:
        return None
    if getOneActorAt(dest) != None:
        return None
    if not isMarbleLocation(dest):
        return None
    if dest.x - start.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x + 1, start.y))
    if start.x - dest.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x - 1, start.y))
    if dest.y - start.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y + 1))
    if start.y - dest.y == 2 and dest.x == start.x:
```

```
        return getOneActorAt(Location(start.x, start.y - 1))

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if loc.x in [0, 1, 5, 6] and loc.y in [0, 1, 5, 6]:
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ". No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ". Marble found")

def dragEvent(e):
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if movingMarble == None:
        return
    destLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble.setLocationOffset(0, 0)
    removeMarble = getRemoveMarble(startLoc, destLoc)
    if removeMarble == None:
        setStatusText("Released at " + str(destLoc) + ". Not a valid move.")
    else:
        removeActor(removeMarble)
        movingMarble.setLocation(destLoc)
        setStatusText("Released at " + str(destLoc)+ ". Marble removed.")


startLoc = None
movingMarble = None

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
    mousePressed = pressEvent, mouseDragged = dragEvent,
    mouseReleased = releaseEvent)
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

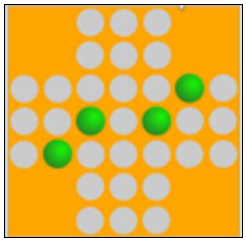
Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Instead of using several early *returns* to leave the function *getRemoveMarble()*, you could also combine the conditions with a Boolean operation. Which programming technique is considered to be more appropriate is a matter of opinion.

CHECKING FOR GAME OVER

Now all that remains is checking if the game is over at the end of each turn. The game is certainly over if only a single marble is left in the game, which means you have achieved the goal of the game.



However, you may forget that there are other game constellations where the game is considered to have ended, namely when there is more than one marble on the board, but you can no longer make a legal turn. It is not certain whether you will ever run into this situation with legal moves, but you have to program ***defensively*** to make sure that you always stay on the safe side. You can expect Murphy's law to also be true for programming: "If anything can go wrong, it goes wrong".

In order to get the situation under control, you can test for each remaining marble individually whether it can be used in a legal move, by implementing a function **isMovePossible()**. There, you check for each marble whether there is a removable intermediary marble in combination with any empty spot [**more...**].

```
from gamegrid import *

def checkGameOver():
    global isGameOver
    marbles = getActors() # get remaining marbles
    if len(marbles) == 1:
        setStatusText("Game over. You won.")
        isGameOver = True
    else:
        # check if there are any valid moves left
        if not isMovePossible():
           setStatusText("Game over. You lost. (No valid moves available)")
           isGameOver = True

def isMovePossible():
   for a in getActors():  # run over all remaining marbles
        for x in range(7): # run over all holes
            for y in range(7):
                loc = Location(x, y)
                if getOneActorAt(loc) == None and \
                  getRemoveMarble(a.getLocation(), Location(x, y)) != None:
                    return True
   return False

def getRemoveMarble(start, dest):
    if getOneActorAt(start) == None:
        return None
    if getOneActorAt(dest) != None:
        return None
    if not isMarbleLocation(dest):
        return None
    if dest.x - start.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x + 1, start.y))
    if start.x - dest.x == 2 and dest.y == start.y:
        return getOneActorAt(Location(start.x - 1, start.y))
    if dest.y - start.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y + 1))
```

```
    if start.y - dest.y == 2 and dest.x == start.x:
        return getOneActorAt(Location(start.x, start.y - 1))
    return None

def isMarbleLocation(loc):
    if loc.x < 0 or loc.x > 6 or loc.y < 0 or loc.y > 6:
        return False
    if loc.x in [0, 1, 5, 6] and loc.y in [0, 1, 5, 6]:
        return False
    return True

def initBoard():
    for x in range(7):
        for y in range(7):
            loc = Location(x, y)
            if isMarbleLocation(loc):
                marble = Actor("sprites/marble.png")
                addActor(marble, loc)
    removeActorsAt(Location(3, 3)) # Remove marble in center

def pressEvent(e):
    global startLoc, movingMarble
    if isGameOver:
        return
    startLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble = getOneActorAt(startLoc)
    if movingMarble == None:
       setStatusText("Pressed at " + str(startLoc) + ".No marble found")
    else:
       setStatusText("Pressed at " + str(startLoc) + ".Marble found")

def dragEvent(e):
    if isGameOver:
        return
    if movingMarble == None:
        return
    startPoint = toPoint(startLoc)
    movingMarble.setLocationOffset(e.getX() - startPoint.x,
                                   e.getY() - startPoint.y)

def releaseEvent(e):
    if isGameOver:
        return
    if movingMarble == None:
        return
    destLoc = toLocationInGrid(e.getX(), e.getY())
    movingMarble.setLocationOffset(0, 0)
    removeMarble = getRemoveMarble(startLoc, destLoc)
    if removeMarble == None:
        setStatusText("Released at " + str(destLoc)
                      + ". Not a valid move.")
    else:
        removeActor(removeMarble)
        movingMarble.setLocation(destLoc)
        setStatusText("Released at " + str(destLoc)+
                      ". Valid move - Marble removed.")
        checkGameOver()


startLoc = None
movingMarble = None
isGameOver = False

makeGameGrid(7, 7, 70, None, "sprites/solitaire_board.png", False,
   mousePressed = pressEvent, mouseDragged = dragEvent,
   mouseReleased = releaseEvent)
setBgColor(Color(255, 166, 0))
setSimulationPeriod(20)
addStatusBar(30)
```

```
setStatusText("Press-drag-release to make a move.")
initBoard()
show()
doRun()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

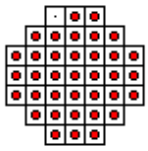
MEMO

After each turn, you check if the game is over using **checkGameOver()**, If it is over, the game is in a very specific state that you can distinguish with the Boolean variable (a flag) **isGameOver = True** .

In particular, you must remember to stop all the mouse actions in *Game Over*. You can do this with an immediate **return** from the mouse callbacks.

EXERCISES

1. Create a French Solitaire board.



2. Expand the Solitaire board with a score that counts and writes out the number of turns. The game should also be able to restart by pressing the space bar.

3. Familiarize yourself with solution strategies of Peg solitaire with the help of a teacher or the Internet [**more...**].

4. Create a Solitaire board from your own imagination.

7.5 SPRITE ANIMATION

INTRODUCTION

When working with the game library *JGameGrid*, all tokens should be derived from the class *Actor* so that they already include many important features and capabilities without any necessary programming effort. However, they get their specific appearance loaded from an image file, called a **sprite**.

Actors in a game are animated in various ways: they move across the game area and change their appearance in the process, e.g. their posture or expression. For this reason, an *Actor* object can be assigned any number of different sprite images that are distinguished by an integer index (the sprite ID). This is simpler than modeling *Actors* with different sprites via class derivation.

Game tokens also often change their place, direction, and rotation angle. The rotation angle should automatically be adjusted to the direction of the movement. In *JGameGrid,* for efficiency reasons, one has to already specify at their definition whether *Actors* can be rotated and which sprite images they are assigned. The latter are, at the creation of the *Actor* object, loaded into an image buffer that also contains the rotated images. At runtime, the images therefore do not have to be loaded from the hard drive or otherwise transformed. By default, 60 sprite images are generated for every 6 degrees of rotation. *JGameGrid* uses an animation concept also available in other game libraries, particularly **Greenfoot** [**more...**] .

Fundamental animation principle:

> The method *act(),* defined for the class *Actor()*, has an **empty** definition part, and so it returns immediately. The user-defined derived actors then override *act()* and thereby implement the specific behavior of the actor.
>
> When adding a character to the game window with *addActor(),* it will be inserted into an **act-order list** (ordered by *Actor* classes). An internal **game loop** (in this case also called a **simulation cycle**) periodically runs through this list and subsequently calls all the actors' specific *act()* methods due to polymorphism.

For this ingenious principle to work, the actors have to be **cooperative**, i.e. *act()* must have **short running code**. Loops and delays have especially catastrophic effects, since other actors must wait for their own call of *act().*

The drawing of the sprite images happens according to the following principle. In the game loop, the images of all *Actors* are copied into a screen buffer according to the order in the **paint-order list** and finally rendered in the game window. The order of execution thus determines the visibility of the sprite images: images of **later** actors cover the ones of all previously drawn actors, they **lie above** them, so to speak. Since the actors are added to the paint-order list when *addActor()* is called, sprites added later will lie above the others. [**more...**]

Although any number of sprite images can be assigned at the initialization of an *Actor*, they **cannot be changed** at runtime.

PROGRAMMING CONCEPTS: *Simulation cycle, cooperative code, factory class, static variable, decoupling*

MOVING A BOW AND SHOOTING ARROWS

You want to shoot arrows that move on a natural trajectory (parabola) using a crossbow that you control with the keyboard. You will use these arrows later to slice flying fruit in half.

You write a class *Crossbow* that is derived from the class *Actor*. When calling the constructor of the base class *Actor, y*ou use *True* to say that it is a rotatable actor. The value 2 indicates that there are 2 sprite images, namely one with a cocked crossbow that has an arrow attached to it and the other for a relaxed crossbow without an arrow. The image files are automatically searched for under the name *sprites/crossbow_0.gif* and *sprites/crossbow_1.gif* and are found in the distribution of *TigerJython*.

```
Actor.__init__(self, True, "sprites/crossbow.gif", 2)
```

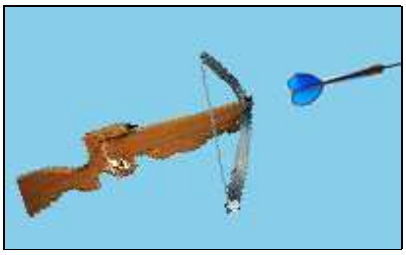
The crossbow is controlled with keyboard events: You can change the direction using the cursor up/down keys and you can shoot the arrow with the spacebar. The callback *keyCallback()* is registered in *makeGameGrid()* as *keyPressed*.

The arrow class *Dart* already gets a bit more complicated, as the arrows have to move on a parabolic trajectory in an x-y coordinate system, with the horizontal x-axis and the vertical y-axis pointing down. The trajectory is not determined by a curve equation, but rather iteratively as a change in the short time *dt*. It is known from kinematics that the new speed coordinates (*vx', vy'*) and the new location coordinates (*px', py'*) after the time difference *dt* are calculated as follows (*g = 9.81m/s^2* is the gravitational acceleration):

$$vx' = vx$$
$$vy' = vy + g * dt$$

$$px' = px + vx * dt$$
$$py' = py + vy * dt$$

You determine the starting values (initial conditions) in the method **reset()**, which is automatically called when you add the *Dart* instance to the game area.



You can give the arrow a new location and direction in *act()*. To save some resources, you remove it from the board as soon as it is outside of the visible window and then bring the crossbow into the firing position again.

```
from gamegrid import *
import math

# ------------------- class Crossbow -----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        self.vx = self.speed * math.cos(math.radians(self.getDirection()))
        self.vy = self.speed * math.sin(math.radians(self.getDirection()))

    def act(self):
        self.vy = self.vy + g * self.dt
```

```
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())

screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

When calling the constructor of the class *Actor* you indicate whether the actor is rotatable and whether it is assigned more than one sprite image. [**more...**]

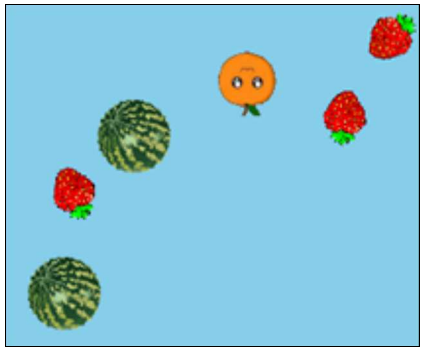
You rotate the direction of the arrow continuously into the direction of the velocity so that it has a natural flight appearance.

FRUIT FACTORY AND MOVING FRUITS

Your program should use three different types of fruits: melons, oranges and strawberries. The fruits are continuously generated in a random order and then move from the upper right edge to the left with a randomly varied horizontal speed on a parabolic trajectory. The three different types of fruit have many similarities and just a few small differences. It would therefore **not be a good idea** to derive the classes *Melon*, *Orange,* and *Strawberry* directly from *Actor*, because you would have to re-implement the shared properties in each class which leads to frowned-upon **duplicated code**. In this situation, it is appropriate to define a helper class *Fruit* where the the similarities can be implemented and where the specific fruits *Melon, Orange,* and *Strawberry* can be derived from..

You delegate the generation of fruit to a type of class called **factory class**. Although it does not have a sprite image, you can (also) derive it from *Actor* so that *act()* can be used to produce new fruits. A *Factory* class has a specific feature: Although it produces multiple fruits, there is only a

single instance [**more...**]. Because of this, it is not common to include a constructor which is intended for the creation of multiple instances. Factory classes therefore have a method called **create()** (or a similarly meaningful name), that creates a single object of the class and returns it as a function value. Each subsequent call of *create()* then merely provides the already created factory instance.



Since the method **create()** is invoked without an instance, it must be **statically** defined with *@staticmethod*.

At the creation of the *FruitFactory*, the maximum number of fruits that the factory can create is specified in the variable *capacity*. Also, each *Actor* can call *setSlowDown()* to slow down the calling frequency of *act()*.

```
from gamegrid import *
import random

# ---------- class Fruit ------------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2) # rotatable, 2 sprites
        self.vx = vx
        self.vy = 0

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt # vx = const
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myFruitFactory = None
    myCapacity = 0
```

```
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
                # slows down act() call for this actor
        return FruitFactory.myFruitFactory

    def act(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random.random() * 20 + 30)
        r = random.randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)

# ------ End of class definitions --------------------

FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
setSimulationPeriod(30)
doRun()
show()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

In a static method, the parameter *self* is not available. Therefore, all variables assigned in *create()* must be **static variables** (the class name is prepended) [**more...**].

Certain functions or methods may still be incompletely coded in a development phase. You can, for example, merely write out to the console that they have been called. You do this here by printing "*Factory expired*". With the adding of actors in the GameGrid using *addActor()*, the image buffer is automatically rendered on the screen so that the actor is immediately visible. As soon as the simulation cycle is started, the rendering happens at every cycle anyway. That is why in this case, you should use **addActorNoRefresh()** since rendering too frequently can cause the screen to flicker.

ASSEMBLING AND DEALING WITH COLLISIONS

The two program parts just written may well have been developed by two research groups. The

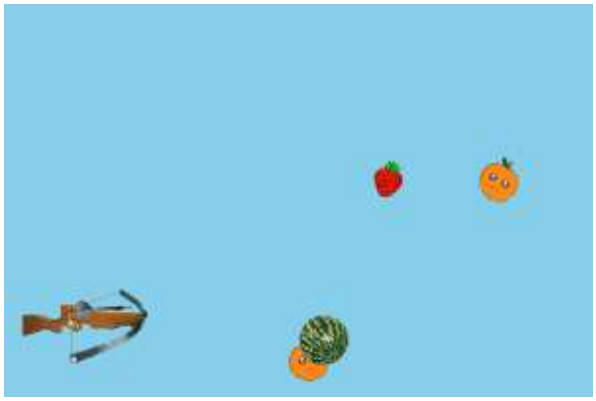
next task is to merge these parts, which is not always easy. However, if the **programming style is consistent** and mostly **decoupled** as it is here, merging the code is significantly easier.

Additionally, you will incorporate a new functionality where the fruits are cut in half when they are hit by an arrow. We have already prepared this, as the fruits have two sprite images: one for the whole fruit and one for the halved fruit.

As you already know, collisions between actors are detected by a collision event. For this, you determine what the possible collision partners are for each actor. Consider the following: when creating an arrow, all currently existing fruits are potential collision partners.

However, do not forget that more fruits are added during the movement of the arrow. That is why you also need to declare all existing arrows (maybe there is only one) as collision partners when creating a fruit.

In *JGameGrid* you can also pass **addCollisionActors()** a whole list of actors as collision partners (more specifically an *ArrayList*). With *getActors(class)* you will get a list with all the actors of the specified class, which you can pass on after converting it to an *ArrayList*.



```
from gamegrid import *
import random
import math

# ---------- class Fruit ------------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            self.removeSelf()
```

```
    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
        return FruitFactory.myFruitFactory

    def act(self):
        self.createRandomFruit()

    def createRandomFruit(self):
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            print "Factory expired"
            return

        vx = -(random.random() * 20 + 30)
        r = random.randint(0, 2)
        if r == 0:
            fruit = Melon(vx)
        elif r == 1:
            fruit = Orange(vx)
        else:
            fruit = Strawberry(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        # for a new fruit, the collision partners are all existing darts
        fruit.addCollisionActors(toArrayList(getActors(Dart)))

# ------------------- class Crossbow ----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
```

```
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()

    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(),
                                crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))


FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Once you have declared the collision partners of your actor with *addCollisionActor()* or *addCollisionActors()*, you have to insert the method *collide()* in the class of the actor which is automatically called at each collision. The return value must be an integer that determines how many simulation cycles collision will now be deactivated (in this case 0). A number greater than 0 is sometimes necessary so that the two partners have time to separate before collisions become active again.

Also note that in **collide(self, actor1, actor2))** *actor1* is the actor of the class, in which *collide()* is defined.

Collision areas are the surrounding rectangles of the sprite image by default (of course they are rotated along with the rotation of the actors). For the dart, you could also set the collision area to a circle around the arrowhead, so that the fruits that collide with the back part of the arrow do not get halved.

```
setCollisionCircle(Point(20, 0), 10)
```

DISPLAYING THE GAME STATE AND DEALING WITH GAME OVER

For dessert, you refine the code by incorporating a game score and user information. The easiest way is to write them out in a status bar.

As you already know, it is favorable to implement a game supervisor in the main part of the program. It should write out the number of the hit and missed fruits and end the game when the fruit factory reaches its capacity. It shows the final score, generates a *Game Over* actor, and prevents the game from continuing on.

```
from gamegrid import *
import random
import math

# ---------- class Fruit -----------------------
class Fruit(Actor):
    def  __init__(self, spriteImg, vx):
        Actor.__init__(self, True, spriteImg, 2)
        self.vx = vx
        self.vy = 0
        self.isSliced = False

    def reset(self): # Called when Fruit is added to GameGrid
        self.px = self.getX()
        self.py = self.getY()

    def act(self):
        self.movePhysically()
        self.turn(10)

    def movePhysically(self):
        self.dt = 0.002 * getSimulationPeriod()
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.cleanUp()

    def cleanUp(self):
        if not self.isInGrid():
            if not self.isSliced:
                FruitFactory.nbMissed += 1
            self.removeSelf()
```

```
    def sliceFruit(self):
        if not self.isSliced:
            self.isSliced = True
            self.show(1)
            FruitFactory.nbHit += 1

    def collide(self, actor1, actor2):
       actor1.sliceFruit()
       return 0

# ------ class Melon -----------
class Melon(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/melon.gif", vx)

# ------ class Orange -----------
class Orange(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/orange.gif", vx)

# ------ class Strawberry -----------
class Strawberry(Fruit):
    def __init__(self, vx):
        Fruit.__init__(self, "sprites/strawberry.gif", vx)

# ------------------- class FruitFactory -------------------
class FruitFactory(Actor):
    myCapacity = 0
    myFruitFactory = None
    nbGenerated = 0
    nbMissed = 0
    nbHit = 0

    @staticmethod
    def create(capacity, slowDown):
        if FruitFactory.myFruitFactory == None:
            FruitFactory.myCapacity = capacity
            FruitFactory.myFruitFactory = FruitFactory()
            FruitFactory.myFruitFactory.setSlowDown(slowDown)
        return FruitFactory.myFruitFactory

    def act(self):
        self.createRandomFruit()

    @staticmethod
    def createRandomFruit():
        if FruitFactory.nbGenerated == FruitFactory.myCapacity:
            return
        vx = -(random.random() * 20 + 30)
        fruitClass = random.choice([Melon, Orange, Strawberry])
        fruit = fruitClass(vx)
        FruitFactory.nbGenerated += 1
        y = int(random.random() * screenHeight / 2)
        addActorNoRefresh(fruit, Location(screenWidth-50, y), 180)
        # for a new fruit, the collision partners are all existing darts
        fruit.addCollisionActors(toArrayList(getActors(Dart)))

# ------------------- class Crossbow ----------------------
class Crossbow(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/crossbow.gif", 2)

# ------ class Dart ----------------
class Dart(Actor):
    def __init__(self, speed):
        Actor.__init__(self, True, "sprites/dart.gif")
        self.speed = speed
        self.dt = 0.005 * getSimulationPeriod()
```

```
    # Called when actor is added to GameGrid
    def reset(self):
        self.px = self.getX()
        self.py = self.getY()
        dx = math.cos(math.radians(self.getDirectionStart()))
        self.vx = self.speed * dx
        dy = math.sin(math.radians(self.getDirectionStart()))
        self.vy = self.speed * dy

    def act(self):
        if isGameOver:
            return
        self.vy = self.vy + g * self.dt
        self.px = self.px + self.vx * self.dt
        self.py = self.py + self.vy * self.dt
        self.setLocation(Location(int(self.px), int(self.py)))
        self.setDirection(math.degrees(math.atan2(self.vy, self.vx)))
        if not self.isInGrid():
            self.removeSelf()
            crossbow.show(0) # Load crossbow

    def collide(self, actor1, actor2):
        actor2.sliceFruit()
        return 0

# ------ End of class definitions --------------------

def keyCallback(e):
    code = e.getKeyCode()
    if code == KeyEvent.VK_UP:
        crossbow.setDirection(crossbow.getDirection() - 5)
    elif code == KeyEvent.VK_DOWN:
        crossbow.setDirection(crossbow.getDirection() + 5)
    elif code == KeyEvent.VK_SPACE:
        if isGameOver:
            return
        if crossbow.getIdVisible() == 1: # Wait until crossbow is loaded
            return
        crossbow.show(1) # crossbow is released
        dart = Dart(100)
        addActorNoRefresh(dart, crossbow.getLocation(), crossbow.getDirection())
        # for a new dart, the collision partners are all existing fruits
        dart.addCollisionActors(toArrayList(getActors(Fruit)))


FACTORY_CAPACITY = 20
FACTORY_SLOWDOWN = 35
screenWidth = 600
screenHeight = 400
g = 9.81
isGameOver = False

makeGameGrid(screenWidth, screenHeight, 1, False, keyPressed = keyCallback)
setTitle("Use Cursor up/down to target, Space to shoot.")
setBgColor(makeColor("skyblue"))
addStatusBar(30)
factory = FruitFactory.create(FACTORY_CAPACITY, FACTORY_SLOWDOWN)
addActor(factory, Location(0, 0))  # needed to run act()
crossbow = Crossbow()
addActor(crossbow, Location(80, 320))
setSimulationPeriod(30)
doRun()
show()

while not isDisposed() and not isGameOver:
   # Don't show message if same
   oldMsg = ""
   msg = "#hit: "+str(FruitFactory.nbHit)+" #missed: "+str(FruitFactory.nbMissed)
   if  msg != oldMsg:
```

```
            setStatusText(msg)
            oldMsg = msg
    if FruitFactory.nbHit + FruitFactory.nbMissed == FACTORY_CAPACITY:
        isGameOver = True
        removeActors(Dart)
        setStatusText("You smashed " + str(FruitFactory.nbHit) + " out of "
        + str(FACTORY_CAPACITY) + " fruits")
        addActor(Actor("sprites/gameover.gif"), Location(300, 200))

    delay(100)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Most user actions should not be allowed at Game Over. The easiest way to implement this is to introduce a flag *isGameOver = True* with which you prohibit the actions using a premature *return* in the corresponding functions and methods.

You should still be allowed to move the crossbow at Game Over, but not shoot.

EXERCISES

1. Count the number of arrows and restrict it to a reasonable maximum number. Once you have used up the specified amount of arrows, the game will also be over. Add appropriate status information too.

2. Add a point score system for the halving of the fruits:

   Melon: 5 points
   Orange: 10 points
   Strawberry: 15 points

3. Make it so that when you press the Enter key after Game Over, the game starts over.

4. Expand or modify the game with some of your own ideas.

Most Important Methods of the Library Class JGameGrid

Module import: from gamegrrid import *

Class GameGrid (global functions when *makeGameGrid()* is called)

Method	Action
GameGrid(nbHorzCells, nbVertCells, cellSize, color)	generates a game window with a given number of horizontal and vertical cells, a given cell size, visible grid lines in a given color, and a navigation bar
GameGrid(nbHorzCells, nbVertCells, cellSize, color, bgImagePath)	generates a game window with a given number of horizontal and vertical cells, a given cell size, grid lines, a background image, and a navigation bar
GameGrid(nbHorzCells, nbVertCells, cellSize, None, bgImagePath, False)	generates a game window with a given number of horizontal and vertical cells, a given cell size, no grid lines, a background image, and no navigation bar
act()	calls periodically after the start of the simulation cycle
addActor(actor, location)	adds the actor to the game window at the given position
addKeyListener(listener)	registers the keyboard listener
addMouseListener(listener, mouseEventMask)	registers the mouse listener
addStatusbar(height)	adds a status window to GameGrid
delay(time)	waits for a set amount of time (in milliseconds)
doPause()	pauses the simulation cycle
doStep()	performs the simulation step by step
doReset()	sets all actors at the starting position and restarts the simulation
doRun()	starts the simulation cycle
getActors(Actor class)	returns all actors of a given class in a list
getBg()	returns the reference to GGBackground
getBgColor()	returns the background color
getKeyCode()	returns the key code of the last pressed key
getOneActorAt(location)	returns the first actor in the given cell (zero if none)
getOneActor(Actor class)	returns the first actor of the given class (zero if none)
getRandomEmptyLocation()	returns a random empty cell location
getRandomLocation()	returns a random cell location
hide()	hides the game window without closing it
isAtBorder(location)	returns *True* if the cell is located at the edge of the game window
isEmpty(location)	returns *True* if the cell is empty
isInGrid(location)	returns *True* if the cell is inside of the game window
kbhit()	returns *True* if a key has been pressed
toLocation(x, y)	returns the cells with the pixel coordinates x and y
openSoundPlayer("wav/ping.wav")	produces a sound file. The following sounds are available in *tigerjython.jar*: bird.wav, boing.wav, cat.wav, click.wav,explore.wav,frog.wav. notify.wav, boing.wav
play()	plays the provided sound
refresh()	refreshes the game window

registerAct(onAct)	registers the callback onAct that is called in every simulation cycle
registerNavigation(started=onStart, stepped=onStep, paused=onPause, resetted = onReset, periodChanged = onPerionChange)	registers the callbacks onStart, onStep, onPause, onReset, onPeriodChange that are called when the navigation bar is visible (not all necessary)
removeActor (actor)	removes an actor from the game window
removeActorsAt(location)	removes all actors located in the specified cell
removeAllActors()	removes all actors from the game window
reset()	puts the defined simulation back to the starting position, with the exception of actors which have already been removed
show()	shows the game window
setBgColor(color)	sets the background color
setSimulationPeriod (milisec)	sets the period of the simulation loop
setStatusTest(text)	sets the text in the status bar
setTitle(text)	sets the title in the window title bar

Class Actor

Actor(spritepath)	generates an actor with the given sprite
Actor(True, spritepath)	generates a rotatable actor with the given sprite
Actor(spritepath, nbSprites)	generates an actor with multiple sprites (index _0, _1,... e.g..fish_0.gif , fish_1.gif,...)
Actor(True, spritepath, nbSprites)	generates a rotatable actor with multiple sprites
act()	calls periodically after the start of the simulation cycle
addActorCollisionListener(listener)	registers the collision listener
addCollisionActor(actor)	registers the collision partner
addMouseTouchListener (listener)	registers the MouseTouchListener
collide(actor1, actor2)	callback when a collision occurs, returns the number of simulation cycles, while the other events are suppressed
getCollisionActors()	returns a list of the collision candidates
getDirection()	returns the direction of movement
getIdVisible()	returns the Id of the visible sprites
getNeighbours(distance)	returns a list of all actors that are the given distance away
getNextMoveLocation (location)	returns the *location* after the next *move()*
getX()	returns the current horizontal cell coordinate
getY()	returns the current vertical cell coordinate
hide()	hides the actor, but does not remove it. After *reset()* it becomes visible again
isInGrid()	returns *True* if the actor is located inside of the game window
isHorzMirror()	returns *True* if the figure is mirrored horizontally
isVertMirror()	returns *True* if the figure is mirrored vertically
isMoveValid()	returns *True* if a *move()* of the actor stays inside of the window
isNearBorder()	returns *True* if the actor is located near the edge of the window
isVisible()	returns *True* if the actor is visible
move()	moves the actor with the current direction into an adjacent cell
move(distance)	moves the actor to the given distance

reset()	called if the actor is added to the GameGrid or if the reset button is pressed
setCollisionCircle (spriteId,center, radius)	sets the circle within the actors which is used for collisions
setCollisionLine(spriteId, startPoint, endPoint)	sets the line within the actors which is used for collisions
setCollisionRectangle(spriteId, center, width, height)	sets the rectangle within the actors which is used for collisions
setCollisionSpot(spriteId, spot)	sets the point within the actors which is used for collisions
setCollisionImage(spriteId)	wählt für die Kollision nicht transparente Pixel . Nur verfügbar, wenn der Partner spot, line oder circle verwendet
setHorzMirror(True)	flips the image horizontally
setVertMirror(True)	flips the image vertically
setSlowDown(factor)	slows the call of the method *act()* for actors with the given factor
setLocation(location)	places the actor in the given cell
setLocationOffset(point)	shifts the middle of the sprite image relative to the center of the cell (location not changed)
setPixelLocation(location)	sets the actor to the given pixel coordinate (location/offset can be customized)
setX(x)	sets the x-coordinate to the specified value
setY(y)	sets the y-coordinate to the specified value
show()	makes the sprite with the ID 0 visible
show(spriteId)	makes the sprite with the specified ID visible
showNextSprite ()	shows the next sprite image (spriteId increases by 1 (modulo nbSprites))
showPreviousSprite()	shows the previous sprite image (spriteId -1 becomes nbSprites - 1)
removeSelf()	removes the actor. After *reset()* it no longer appears
reset()	is called by GameGrid.addActor() and when the reset button is pressed
turn(angle)	changes the direction of movement by the given angle (in degrees clockwise)

Class Location

Location(x, y)	generates a location object with the given horizontal and vertical cell coordinates
Location(location)	generates a location object with the given location (clone)
clone()	returns the new location with the same coordinates
equals(location)	returns *True* if the current location is identical to the one given above
get4CompassDirectionTo(location)	returns a list with 4 adjacent locations (WEST, EAST, NORTH, SOUTH)
getCompassDirectionTo(location)	returns a list of 8 neighboring locations (also diagonally)
getDirectionTo(location)	returns the direction of the current to the given position in degrees (0 degrees = east)
getNeighbourLocation(direction)	returns one of the 8 neighboring cells. It returns the cell that is closest to the given direction

getNeighbourLocations(distance)	returns a list of all the cells with centers inside of the given distance
getX()	returns the current horizontal cell coordinate
getY()	returns the current vertical cell coordinate

Class GGBackground

clear()	clears the background and fills it with the current background color
clear(color)	clears the background and fills it with the given background color
drawCircle(center, radius)	draws a circle with the given center and radius (pixel coordinates)
drawLine(x1,y1, x2, y2)	draws a line with the given end points (pixel coordinates)
drawLine(pt1, pt2)	draws a line with the given end points (pixel coordinates)
drawPoint(pt)	draws a point (pixel coordinates)
drawRectangle(pt1, pt2)	draws a rectangle with the given diagonal vertices (pixel coordinates)
drawText(text, pt)	writes text to the position with the given starting point (pixel coordinates)
fillCell(location, color)	fills the given cell with the given color (pixel coordinates)
fillCircle(center,radius)	draws a filled circle with the given center and radius (pixel coordinates)
getBgColor()	returns the current background color
getColor(location)	returns the background color in the center of a cell (actors are not taken into account)
save()	saves the current background
setBgColor(color)	changes the background color
setFont(font)	sets the font
setLineWidth(width)	sets the line width
setPaintColor(color)	sets the drawing color
setPaintMode()	draws regardless of the existing background
setXORMode(color)	the second drawing produces a background again??
restore()	restores the previously saved background

chapter eight

COMPUTER EXPERIMENTS

Learning Objectives

- You can solve simple stochastic problems with a computer simulation using random numbers.
- You understand that random experiments are subject to statistical fluctuations, you can represent results as a frequency distribution and interpret them.
- You can examine the significance of a sample using a computer simulation and you know the concept of chi-square tests.
- You know how to use the computer to simulate populations.
- You know what the Mandelbrot set is and how to represent it graphically.
- You know what fundamentals and overtones are and you know the concept of a spectrum.

"I only believe in statistics that I doctored myself."

Attributed to Winston Churchill

8.1 SIMULATIONS

INTRODUCTION

Computer simulations do not only play an important role in research and in the industry, but also also in the finance world. They are used to simulate the behavior of a real system using a computer. Computer simulations have the advantage of being inexpensive and environmentally friendly, as well as safe, compared to real experiments and studies. However, they can usually never reflect reality with full accuracy. There are many reasons for this:

- reality can never be perfectly represented by numbers due to errors in measurement (except for in enumerations)
- often the interaction of the components is not precisely known, since either the underlying laws are not exact [more...] or not all influences are taken into account [more...]

Nevertheless, computer simulations are becoming more and more precise with increasing computational power, just think of the weather forecasts for the next few days.

Chance plays an exceptionally big role in our lives, as we make many decisions based on an intuitive assessment of probabilities and not just on the basis of purely logical arguments. However, the use of chance can also greatly simplify problems with exact solutions. One example is that it can be very time consuming to exactly determine the shortest possible path from A to B on a road map with many different connection possibilities using an algorithm; it is sufficient for practical use to find the most probably shortest possible path [more...]

PROGRAMMING CONCEPTS: *Computer simulation. computer experiment, statistical fluctuations*

THE COMPUTER AS A GAME PARTNER

Your companion Nora suggests the following game: "You can throw three dice. If you roll a six, you win and I'll give you a marble. If you don't roll a six, I win and you have to give me a marble".

At first glance the game appears to be fair because you think about it quickly and realize that for each die, the probability of rolling a six is 1/6, and so the chance to roll a six in the first, second, or third turn is 1/6 + 1/6 + 1/6 = 1/2.

You can verify this thought process with the computer and your programming skills. You thereby assume that it does not matter whether you roll the 3 dice consecutively or all at the same time. So in other words, the probability of a die to obtain a certain number is independent of the other dice and it is always 1/6.
There are two ways to tackle the problem, either **statistically** or **combinatorially**. The statistical solution corresponds to the real game. You simulate the throwing of the dice by **repeatedly** generating 3 random numbers between 1 and 6 and then counting the winning cases.

```
from random import randint

n = 1000 # number of games
won = 0
repeat n:
    a = randint(1, 6)
    b = randint(1, 6)
    c = randint(1, 6)
    if a == 6 or b == 6 or c == 6:
        won += 1

print "Won:", won, " of ", n, "games"
print "My winning percentage:", won / n
```

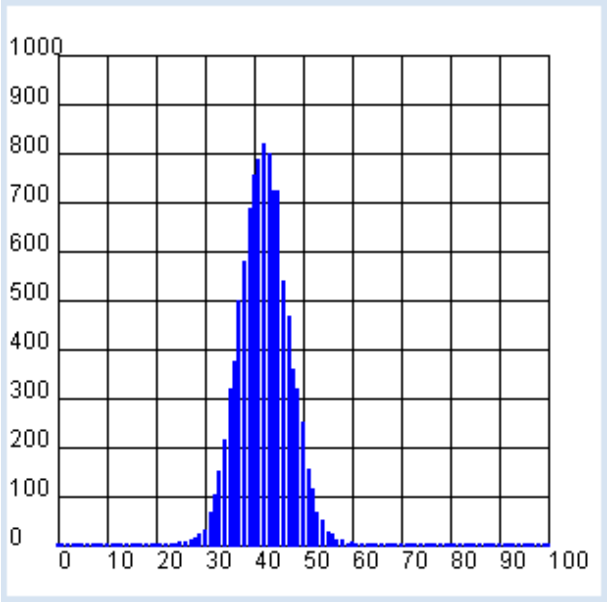
Highlight program code (Ctrl+C copy, Ctrl+V paste)

The result is a winning percentage of about 0.42, not 0.5 as you expected. The value easily changes from simulation to simulation though, because it is subject to **statistical fluctuations**. As you might intuitively expect, the result is **more accurate** the **more tests** you do.

Statistical fluctuations are of great importance in computer simulations.

In order to examine them, you conduct the experiment with purposely few games (let's say 100) many times (let's say 10000 times) and draw a **frequency diagram** of the games won. It results in an interesting bell-shaped **distribution,** typical for statistics.

You use a *GPanel* as a graphics window in the program. You can also display a coordinate grid using drawGrid(). Implement a single hundred-times test with the function sim(), which returns the number of games won whose fluctuations you want to investigate.




```
from gpanel import *
from random import randint

z = 10000
n = 100

def sim():
    won = 0
    repeat n:
        a = randint(1, 6)
        b = randint(1, 6)
        c = randint(1, 6)
        if a == 6 or b == 6 or c == 6:
            won += 1
    return won

makeGPanel(-10, 110, -100, 1100)
drawGrid(0, 100, 0, 1000)
h = [0] * (n + 1)
title("Simulation started. Please wait...")
```

```
repeat z:
    x = sim()
    h[x] += 1
title("Simulation ended")

lineWidth(2)
setColor("blue")
for x in range(n + 1):
    line(x, 0, x, h[x])
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The maximum of the distribution is approximately at 42, since the probability of winning is around 0.42 and you play 100 games each time. If you play 100 times with Nora it is possible that you win the game over 50 times, despite your only 0.42 chance of winning. However, the probability for this is quite low (ca. 5 %) and therefore the game is not fair. Computer experiments with random numbers are subject to statistical fluctuations that get smaller the larger number of attempts.

For the combinatorial solution, you let the computer run all possible rolls with 3 dice one after another. The first, second, and third roll can each result in a number from 1 to 6. In the nested for loop you form all triples of numbers and count the total possibilities with the variable *possible,* whereas you count your winning cases which contain at least one 6 with the variable *favorable*.

```
possible = 0
favorable = 0
for i in range(1, 7):
    for j in range(1, 7):
        for k in range(1, 7):
            possible += 1
            if i == 6 or j == 6 or k == 6:
                favorable += 1
print "favorable:", favorable, "possible:", possible
print "My winning percentage:", guenstige / possible
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

This results in 91 favorable of 216 possible cases and thus a winning probability *w = favorable / possible* of 91/216 = 0.42, which you also get with the computer simulation.

ADDITIONAL MATERIAL

You could, of course, also solve this problem entirely without a computer. For this, think of the following: There are three possible winning events E1, E2, E3:

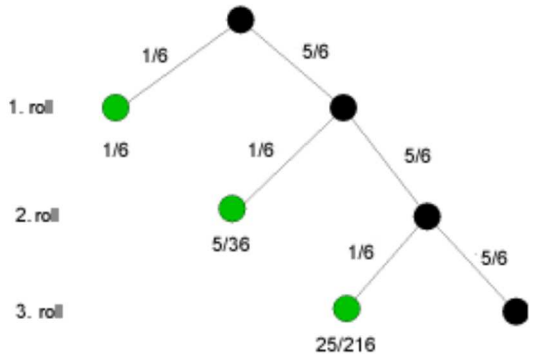
E1: Getting a 6 on the first roll. Probability: 1/6

E2: No 6 in the first roll, but a 6 in the second round.
Probability: 5 /6 * 1/6

E3: No 6 in either the first or second roll, but a 6 in the third round.
Probability: 5/6 * 5 /6 * 1/6

Since E1, E2, and E3 are independent of each other, the probability is the sum, i.e. 1 /6 + 5 /36 + 25 /216 = 91/216 = 0.421296.

You can display the process as a tree:



There is also an *ideal way* to get the solution: the probability of rolling no 6's at all is p = 5/6 * 5/6 * 5/6 = 125/216. Therefore, the desired probability is w = 1 - p = 91/216.

EXERCISES

1. The Duke Ferdinando de Medici of Florenz determined in the year 1600 that when throwing 3 dice a total of 9 or 10 pips (the dots on the die) can be obtained with a same number of possibilities:

Sum of pips 9	**Sum of pips 10**
1 + 2 + 6	1 + 3 + 6
1 + 3 + 5	1 + 4 + 5
2 + 2 + 5	2 + 2 + 6
2 + 3 + 4	2 + 3 + 5
3 + 3 + 3	2 + 4 + 4

   The Duke found, however, that rolling the totals of 9 and 10 are not equally probable and he asked mathematics professor Galileo Galilei for advice. Calculate these probabilities with a computer simulation in two ways:
   a. statistically
   b. combinatorially

2. Using a statistical computer simulation, determine the probability that at least two children have the same birthday (no leap year) in a class of (at least) 20 children.

3. In 1650 in Paris, the Chevalier de Méré asked the mathematician Blaise Pascal about the odds of the following two events:
   a. rolling at least one 6 after 4 rolls
   b. rolling at least one double 6 after 24 rolls.
   He believed that the odds of winning are equal, since even though with b) the probability of winning is 6 times as low, there 6 times as many tries. Was he right?

4*. In the game with Nora, determine how high the probability is that you win more than 50 times in a game consisting of 100 rolls.

8.2 POPULATIONS

INTRODUCTION

Computer simulations are often used to make predictions about the behavior of a system in the future, based on time observation or a certain time span in the recent past. Such predictions can be of great strategic significance and can prompt us, for example, to rethink early enough in a scenario leading to a catastrophe. Hot topics today are the prediction of the global climate and population growth.

We understand a population as a system of individuals whose number changes as a result of internal mechanisms, interactions, and external influences over the course of time. If external influences are disregarded, we speak of a closed system. For many populations the change in population size is proportional to the current size of the population. The change of the current value is calculated from the growth rate as follows:

*new value - old value = old value * growth rate * time interval*

Because the left shows the difference of the new value from the old value, this relationship is called a **difference equation**. The growth rate can also be interpreted as an increase probability per individual and time unit. If it is negative, it decreases the size of the population. The growth rate may well change over the course of time.

PROGRAMMING CONCEPTS: *Difference equation, growth rate, exponential/limited growth, life table, population pyramid, predator-prey system*

EXPONENTIAL GROWTH

Population projections are of great interest and can massively affect the political decision-making process. The latest example is that of the debate going on about the regulation of the proportion of foreigners in the population.

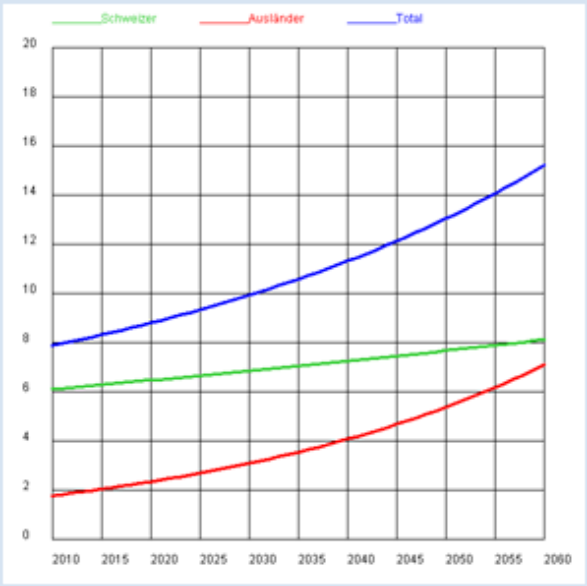
You can find the number of inhabitants in Switzerland each year for the years 2010 and 2011 from the Swiss Federal Statistical Office (source:: **http://www.bfs.admin.ch**, keyword: STAT-TAB):

2010: Total z_0 = 7 870 134, of which s_0 = 6 103 857 are Swiss
2011: Total z_1 = 7 954 662, of which s_1 = 6 138 668 are Swiss

Can you create a forecast of the proportion of foreigners for the next 50 years from this information? You should first calculate the number of foreigners using the numbers $a_0 = z_0 - s_0$ and $a_1 = z_1 - s_0$ and from this the annual growth rate between 2010 and 2011 for Swiss and foreigners.

$$rs = \frac{s_1 - s_0}{s_0} = 0.57\% \qquad \text{bzw.} \qquad ra = \frac{a_1 - a_0}{a_0} = 2.81\%$$

It should now be easy for you to investigate the composition of the population for the next 50 years, **provided that these growth rates remain the same**. You can do this with a calculator, a spreadsheet program, or with *Python*. You can visualize the calculated values in a graph.




```
from gpanel import *

# source: Swiss Federal Statistical Office, STAT-TAB
z2010 = 7870134 # Total 2010
z2011 = 7954662 # Total 2011
s2010 = 6103857 # Swiss 2010
s2011 = 6138668 # Swiss 2011

def drawGrid():
    # Horizontal
    for i in range(11):
        y = 2000000 * i
        line(0, y, 50, y)
        text(-3, y, str(2 * i))
    # Vertical
    for k in range(11):
        x = 5 * k
        line(x, 0, x, 20000000)
        text(x, -1000000, str(int(x + 2010)))

def drawLegend():
    setColor("lime green")
    y = 21000000
    move(0, y)
    draw(5, y)
    text("Swiss")
    setColor("red")
    move(15, y)
    draw(20, y)
    text("foreigner")
    setColor("blue")
    move(30, y)
    draw(35, y)
    text("Total")

makeGPanel(-5, 55, -2000000, 22000000)
title("Population growth extended")
drawGrid()
drawLegend()

a2010 = z2010 - s2010 # foreigners 2010
a2011 = z2011 - s2011 # foreigners 2011

lineWidth(3)
setColor("blue")
line(0, z2010, 1, z2011)
setColor("lime green")
```

```
line(0, s2010, 1, s2011)
setColor("red")
line(0, a2010, 1, a2011)

rs = (s2011 - s2010) / s2010  # Swiss growth rate
ra = (a2011 - a2010) / a2010  # foreigners growth rate

# iteration
s = s2011
a = a2011
z = s + a
sOld = s
aOld = a
zOld = z
for i in range(0, 49):
    s = s + rs * s  # model assumptions
    a = a + ra * a  # model assumptions
    z = s + a
    setColor("blue")
    line(i + 1, zOld, i + 2, z)
    setColor("lime green")
    line(i + 1, sOld, i + 2, s)
    setColor("red")
    line(i + 1, aOld, i + 2, a)
    zOld = z
    sOld = s
    aOld = a
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

As you can gather from the figures, the proportion of foreigners doubles from 2010 to 2035, so just within 25 years, and in another 25 years it quadruples. The population size obviously increases proportionally to the constant growth rate. If T is the doubling time, the population size y after time t for an initial size A is apparently:

$$y = A * 2^{t/T}$$

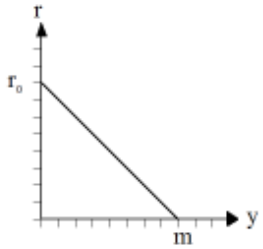
Since time is in the exponent, this rapid growth is called an **exponential growth**.

LIMITED GROWTH

Many populations reside in an environment with limited resources. The rapid exponential increase with a constant growth rate r is therefore bounded. Already about 100 years ago the biologist Carlson determined the following quantities (mg) for a yeast bacteria culture after each hour in an experiment:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
9.6	18.3	29.0	47.2	71.1	119.1	174.6	257.3	350.7	441.0	513.3	559.7	594.8	629.4	640.8	651.1	655.9	659.8	661.8

You can understand the experimental process with a model where the exponential growth experiences saturation. For this you can let the growth rate decrease linearly with an increasing population size *y* until it is zero at a certain **saturation value** *m*.



As you can easily verify by substituting y = 0 and y = m, we get the following formula:

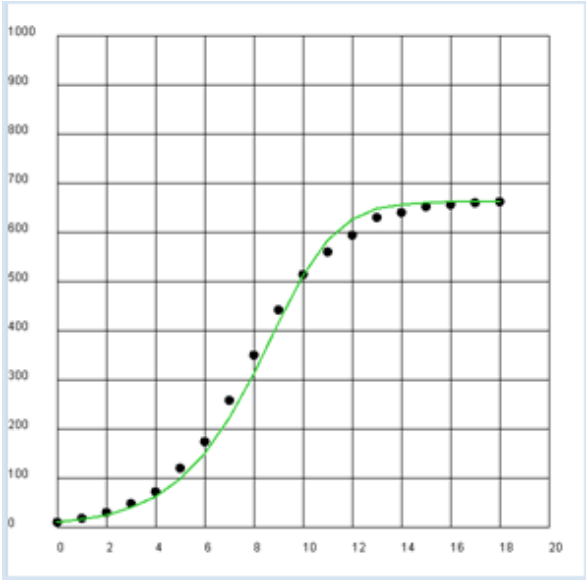
$$r = r_0 * (1 - \frac{y}{m}) = \frac{r_0}{m} * (m - y)$$

With this assumption, you can graphically display the temporal process and also draw the experimental values in a short program. To do this, repeat the difference equation with:

dy: new value - old value
y: old value
dt: time increment
λ: growth rate

so you can write:

$$dy = y * r * dt = y * r_0 * (1 - \frac{r_0}{m}) * dt$$



Using the initial value y_0 = 9.6 mg, the saturation quantity m = 662 mg and the initial growth rate r_0 = 0.62 /h we obtain a good correlation between theory and experiment.

```
from gpanel import *

z = [9.6, 18.3, 29.0, 47.2, 71.1, 119.1, 174.6, 257.3, 350.7, 441.0, 513.3,
559.7, 594.8, 629.4, 640.8, 651.1, 655.9, 659.6, 661.8]

def r(y):
    return r0 * (1 - y / m)

r0 = 0.62
```

```
y = 9.6
m = 662

makeGPanel(-2, 22, -100, 1100)
title("Bacterial growth")
drawGrid(0, 20, 0, 1000)
lineWidth(2)
for n in range(0, 19):
    move(n, z[n])
    setColor("black")
    fillCircle(0.2)
    if n > 0:
        dy = y * r(y)
        yNew = y + dy
        setColor("lime green")
        line(n - 1, y, n, yNew)
        y = yNew
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

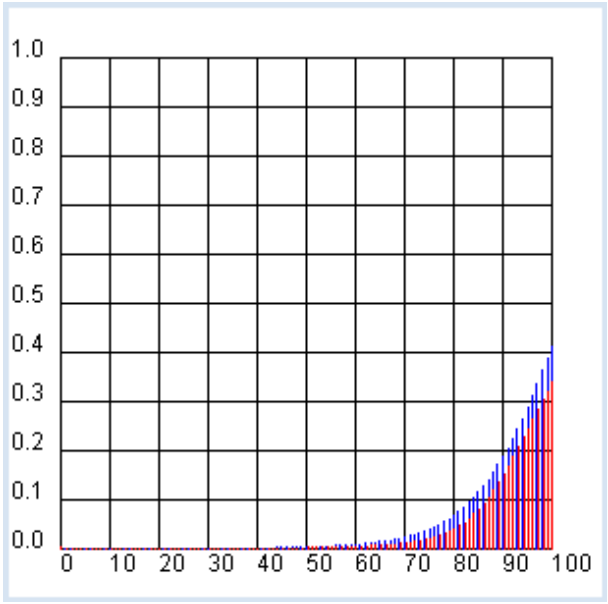
MEMO

Assuming a linear decrease of the growth rate results in an "S" shaped saturation curve typical of the population size (also called **logistic growth** or sigmoid curve).

LIFE TABLES

A possible way to measure the health of a population is to look at the probability of surviving a certain age or of dying at an old age, respectively. If you wish to analyze the age distribution of the Swiss population, you can use current data from the Federal Statistical Office again, namely the so-called *life tables* (source: **http://www.bfs.admin.ch,** keyword: STAT-TAB). These tables contain the observed probabilities (*qx* and *qy*) for men and women to die at a certain age, separated by gender. Their determination is basically simple: one considers all deaths in the past year separately for men and women and calculates the frequency of 0-year-olds (people who died between birth and one year of age), 1-year-olds, etc. Afterwards, one divides each number by the total number in the corresponding age group at the beginning of the year.

(You can create an Excel table for STAT-TAB and copy the columns for *qx* and *qy* into text files *qx.dat* or *qy.dat*, or just download the files from **here**. Copy them into the directory where your program is located.) Input the data into the program in a list *qx* or *qy*. Since the numbers sometimes contain spaces or apostrophes for better readability, you must remove them. First, you simply create a graphical representation of the read data.



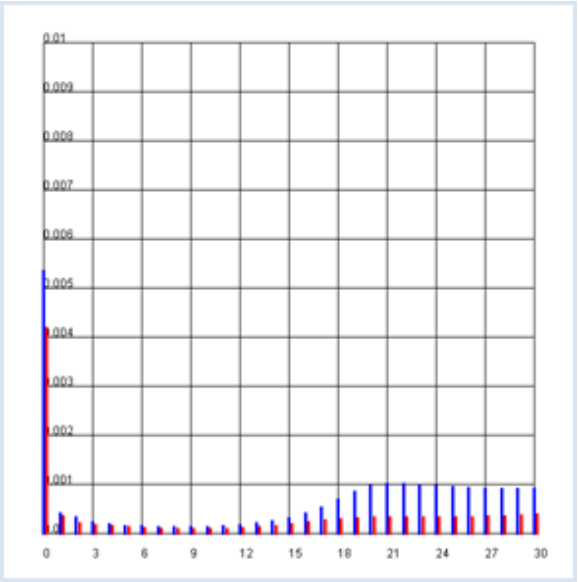
```
import exceptions
from gpanel import *

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

makeGPanel(-10, 110, -0.1, 1.1)
title("Mortality probability (blue -> male, red  -> female)")
drawGrid(0, 100, 0, 1.0)
qx = readData("qx.dat")
qy = readData("qy.dat")
for t in range(101):
    setColor("blue")
    p = qx[t]
    line(t, 0, t, p)
    setColor("red")
    q = qy[t]
    line(t + 0.2, 0, t + 0.2, q)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

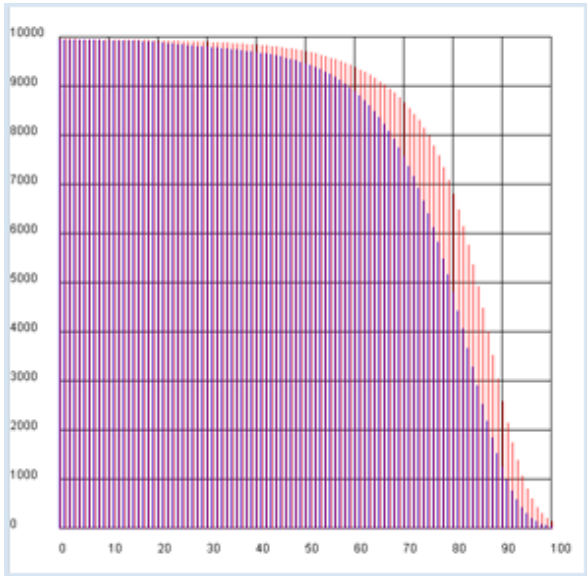
The curve clearly shows that on average, women live longer than men. The course in the first 30 years of life is also interesting.

Significantly more boys than girls die in the first year of life, as well as during the ages from 15- to 30-years old. Come up with some of your own thoughts about this graph.



TEMPORAL EVOLUTION OF A POPULATION

With the help of the life table and a computer program, you can tackle many interesting demographic questions in a scientifically correct way. In this example you examine how a population of 10,000 newborns will evolve over the next 100 years. In doing so, use the values *qx* and *qy* as negative growth rates.




```
import exceptions
from gpanel import *

n = 10000 # size of the population

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

makeGPanel(-10, 110, -1000, 11000)
title("Population behavior/predictions  (blue -> male, red -> female)")
drawGrid(0, 100, 0, 10000)
qx = readData("qx.dat")
qy = readData("qy.dat")
x = n # males
y = n # females
for t in range(101):
    setColor("blue")
    rx = qx[t]
    x = x - x * rx
    line(t, 0, t, x)
    setColor("red")
    ry = qy[t]
    y = y - y * ry
    line(t + 0.2, 0, t + 0.2, y)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

LIFE EXPECTANCY OF WOMEN AND MEN

In the previous analysis it became clear once again that women live longer than men. You can also express this difference with a single quantity called the **life expectancy**. This is the average achieved age of either women or men.

Briefly recall how an average, for example the average grade of a school class, is defined: You calculate the sum *s* of the grades of all students and divide it by the number of students *n*. If you are looking for simplicity, you can assume that only the integer grades between 1-6 occur and so you can calculate *s* as follows:

s = number of students with the grade 1 * 1 + number of students with the grade 2 * 2 + ... number of students with the grade 6 * 6

or more generally:

average = sum of (frequency of the value * value) divided by the total number

If you read the frequencies from a frequency distribution *h* of the values *x* (in this case, the grades 1 to 6), one also calls the average the **expected value** and you can write

$$E = \frac{x_1 * h_1 + x_2 * h_2 + \ldots + x_n * h_n}{h_1 + h_2 + \ldots + h_n}$$

As you can see, the frequencie h_i are **weighted** with their value x_i in the sum.

Life expectancy is nothing else than the expected value for the age at which women and men die. In order to calculate it with a computer simulation, you begin with a certain amount of men and women (n = 10000) and determine the number of men (*hx*) or women (*hy*) that die between the ages t and t + 1. Evidently these numbers can be expressed as follows, using the size of the population *x* and *y* at the time *t* which you calculated in the previous program and the death rates *rx* and *ry*:

$$hx = x * rx \quad \text{bzw. } hy = y * ry$$

```
n = 10000 # size of the population

def readData(filename):
    table = []
    fData = open(filename)

    while True:
        line = fData.readline().replace(" ", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

qx = readData("qx.dat")
qy = readData("qy.dat")
x = n
y = n
xSum = 0
ySum = 0
for t in range(101):
    rx = qx[t]
    x = x - x * rx
    mx = x * rx # male deaths
```

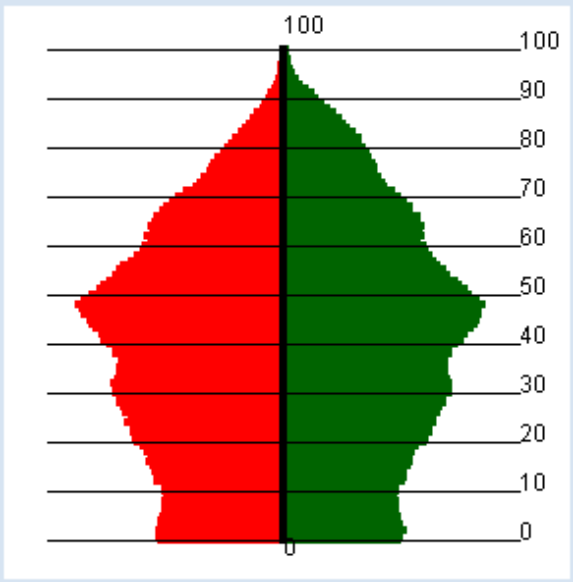
```
    xSum = xSum + mx * t # male sum
    ry = qy[t]
    y = y - y * ry
    my = y * ry # female deaths
    ySum = ySum + my * t # female sum

print "Male life expectancy:", xSum / 10000
print "Female life expectancy:", ySum / 10000
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

The data of the Swiss population yields a life expectancy of men of about 76 years of women about 81 years.

POPULATION PYRAMID

In demographic studies, the population is often grouped by year and from this, a frequency diagram is created. If you have two groups that you would like to compare, you can plot the frequencies of one group to the left and the ones of the other group to the right. Using this method for comparing women and men results in a beautiful pyramid-like graphic.



You can again take the current data (31. December 2012) from a table that you can find on the Swiss Federal Statistical Office website (**http://www.bfs.admin.ch**, keyword: STAT-TAB) and copy them from the Excel table into the test files *zx.dat* and *zy.dat*. You can also download them **hier**.

```
import exceptions
from gpanel import *

def readData(filename):
    table = []
    fData = open(filename)
    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

def drawAxis():
    text(0, -3, "0")
    line(0, 0, 0, 100)
    text(0, 103, "100")

makeGPanel(-100000, 100000, -10, 110)
title("Population pyramid (green -> male, red ->  female)")
lineWidth(4)
zx = readData("zx.dat")
```

```
zy = readData("zy.dat")
for t in range(101):
    setColor("red")
    x = zx[t]
    line(0, t, -x, t)
    setColor("darkgreen")
    y = zy[t]
    line(0, t, y, t)
setColor("black")
drawAxis()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

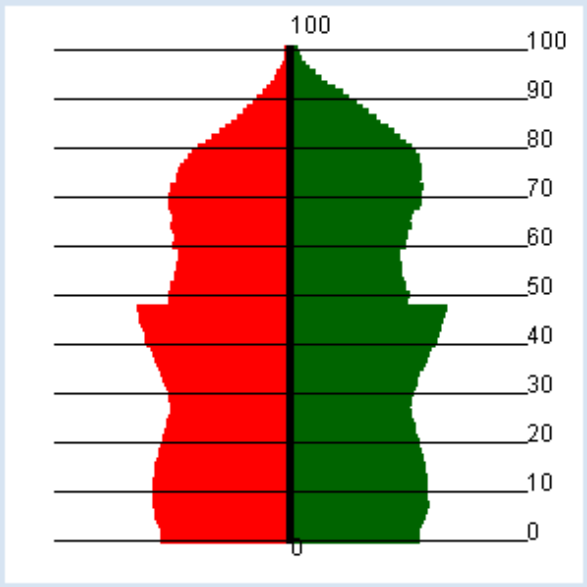
MEMO

It is easy to spot the baby boomers born in the years 1955 – 1965 (47 - 57 years old).

CHANGE OF THE AGE DISTRIBUTION

An analysis of how the age distribution changes over decades can expose important information, such as how a society changes. You can simulate the current age distribution for the next 100 years under the following conditions:

- There is no immigration or migration from the outside (closed society)
- Deaths are taken into account according to the mortality tables
- Each woman of childbearing age from 20 to 39 will have a certain number of children *k* (girls and boys are equally likely). At the moment we will assume k = 2.



With a **key press** you can always insert a year?

```
import exceptions
from gpanel import *

k = 2.0

def readData(filename):
    table = []
    fData = open(filename)
    while True:
        line = fData.readline().replace(" ", "").replace("'", "")
        if line == "":
            break
        line = line[:-1] # remove trailing \n
        try:
            q = float(line)
        except exceptions.ValueError:
           break
        table.append(q)
    fData.close()
    return table

def drawAxis():
```

```
    text(0, -3, "0")
    line(0, 0, 0, 100)
    text(0, 103, "100")
    lineWidth(1)
    for y in range(11):
        line(-80000, 10* y, 80000, 10 * y)
        text(str(10 * y))

def drawPyramid():
    clear()
    title("Number of children: " + str(k) + ", year: " + str(year) +
          ", total population: " + str(getTotal()))
    lineWidth(4)
    for t in range(101):
        setColor("red")
        x = zx[t]
        line(0, t, -x, t)
        setColor("darkgreen")
        y = zy[t]
        line(0, t, y, t)
    setColor("black")
    drawAxis()
    repaint()

def getTotal():
    total = 0
    for t in range(101):
        total += zx[t] + zy[t]
    return int(total)

def updatePop():
    global zx, zy
    zxnew = [0] * 110
    zynew = [0] * 110
    # getting older and dying
    for t in range(101):
        zxnew[t + 1] = zx[t] - zx[t] * qx[t]
        zynew[t + 1] = zy[t] - zy[t] * qy[t]
    # making a baby
    r = k / 20
    nbMother = 0
    for t in range(20, 40):
        nbMother += zy[t]
    zxnew[0] = r / 2 * nbMother
    zynew[0] = zxnew[0]
    zx = zxnew
    zy = zynew

makeGPanel(-100000, 100000, -10, 110)
zx = readData("zx.dat")
zy = readData("zy.dat")
qx = readData("qx.dat")
qy = readData("qy.dat")
year = 2012
enableRepaint(False)
while True:
    drawPyramid()
    getKeyWait()
    year += 1
    updatePop()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

It turns out that the future of the population is very sensitively dependent on the number *k*. Even with the value *k = 2*, the population decreases in the long term. To stop the screen from flickering when you press a key, you should disable automatic rendering with **enableRepaint(False)**. In **drawPyramid()** the graphics are then merely deleted from the backing storage (offscreen buffer), and are only newly rendered to the screen after the recalculation of **repaint()**.

EXERCISES

1. A population consists of 2 individuals at time 0. Each year it increases at a birth rate of 10% (number of births per year per individual). Simulate this for the first 100 years (display it graphically as a bar graph).

2a. In a non-aging population, the mortality probability always remains the same regardless of age. There is no such population for living beings, but radioactive atoms (radionuclides) behave exactly this way. Instead of calling this mortality probability, we call it **decay probability**. Simulate a population of 10,000 radionuclides whose decay probability amounts to 0.1 for the first 100 years (display it graphically as a bar graph).

2b. In the diagram, draw the times at which the population has shrunk to approximately 1/2, 1/4, 1/8, and 1/16 of the initial size, as vertical lines. What can you guess?

2c* Radioactive decay takes place according to the following law:

$$N = N_0 * e^{-\lambda t}$$

N_0: number of radionuclides at the time t = 0
N: number of radionuclides at the time *t*
λ: decay probability per time unit (decay constant)

Enter the best possible adapted curve shape using the color red in the graphic from 2a.

3. Life expectancy can also be calculated by a statistical computer simulation. To do this, you simulate the life of a single individual from year to year. Let the computer choose a random number between 0 and 1 and allow the individual to die if the number is less than the mortality probability *q*. You then add up the achieved life duration. Once you have performed this simulation for 10,000 individuals, divide the total by 10,000. Determine the life expectancy of a female using this method with the values from *qy.dat*.

ADDITIONAL MATERIAL

PREDATOR-PREY SYSTEMS

The behavior of two populations in a particular ecosystem which affect each other is very interesting. Assume the following scenario:
Bunnies and foxes reside in a closed territory. The bunnies multiply at a constant growth rate *rx*. If a fox meets a bunny, there is a certain probability that the fox will snatch it. In turn, the foxes die with a mortality rate *ry*. Their growth rate is determined by the consumption of bunnies.

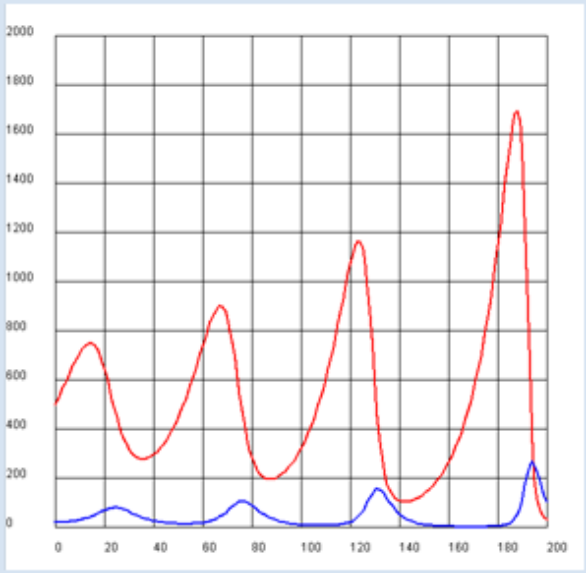
If you assume that the probability of the foxes and bunnies meeting is equal to the product of the number of bunnies *x* and foxes *y*, there are two difference equations for *x* and *y* [**more...**].

$xNew - x = rx * x - gx * x * y$
$yNew - y = -ry * y + gy * x * y$

Use the following values:

rx = 0.08
ry = 0.2
gx = 0.002
gy = 0.0004

and use the initial populations x = 500 bunnies and y = 20 foxes. For now, perform the simulation for 200 generations.




```
from gpanel import *

rx = 0.08
ry = 0.2
gx = 0.002
gy = 0.0004

def dx():
    return rx * x - gx * x * y

def dy():
    return -ry * y + gy * x * y

x = 500
y = 20

makeGPanel(-20, 220, -200, 2200)
title("Predator-Prey system (red: bunnies, blue: foxes)")
drawGrid(0, 200, 0, 2000)
lineWidth(2)
for n in range(200):
    xNew = x + dx()
    yNew = y + dy()
    setColor("red")
    line(n, x, n + 1, xNew)
    setColor("blue")
    line(n, y, n + 1, yNew)
    x = xNew
    y = yNew
```

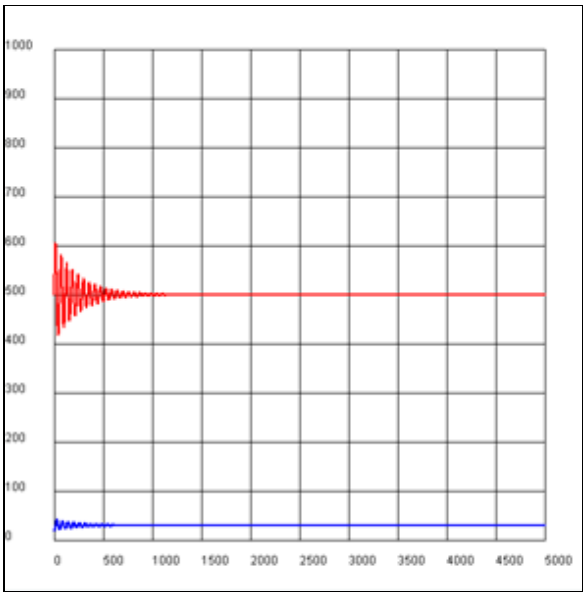
Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

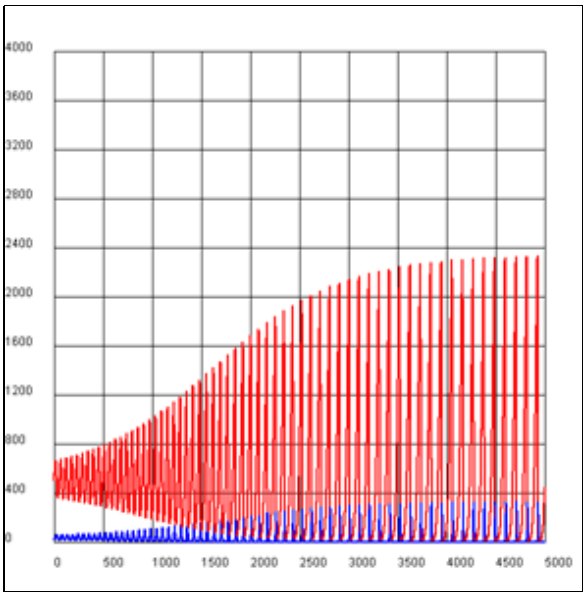
The number of the bunnies and foxes consistently fluctuates up and down. Qualitatively, this cycle is understood as follows: since the foxes eat the bunnies, they multiply particularly strongly, when there are many bunnies. Since this in turn depletes the population of the bunnies, the breeding of foxes slows down. It is during this time that the number of bunnies increases again (even beyond any limits).

EXERCISES

1. Implement a boundary of the habitat for the bunnies according to the logistic growth with a growth rate $rx' = rx(1 - x/m)$ with otherwise identical values as you did above. Show that when m = 2000 the oscillation decays over time, whereas when m = 3500 it oscillates regularly.

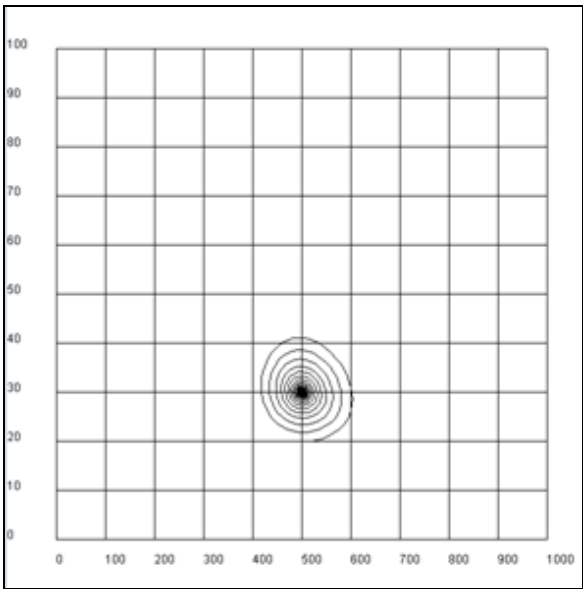


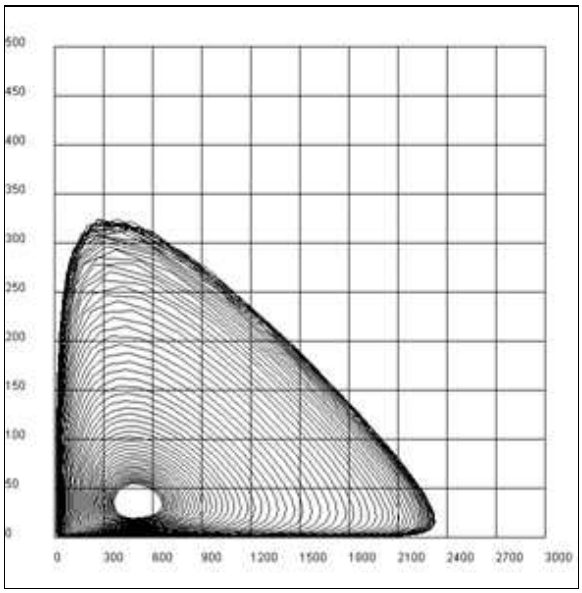
m = 2000, oscillation decays/fades away



m = 3500, oscillation is stable

2. A chart/graph where the sizes of the population are plotted against each other is called a **phase diagram**. Write a program that draws the phase diagram for the two cases from exercise 1. Do you understand the behavior?





8.3 HYPOTHESES, STATISTICAL TESTS

INTRODUCTION

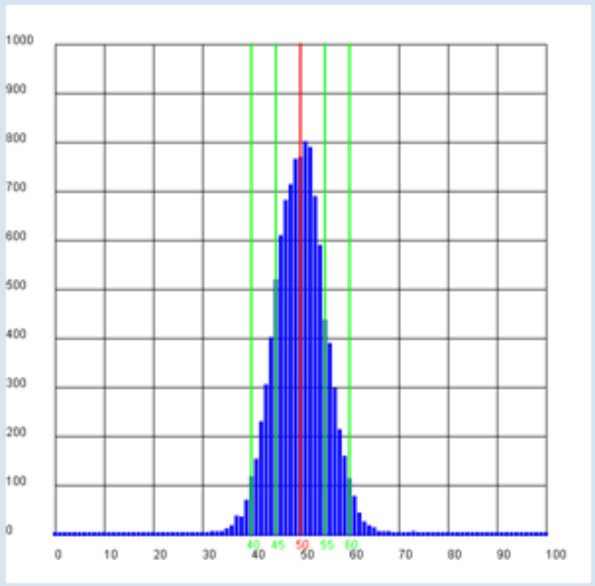
You make a hypothesis (called the **null hypothesis**), for example to check if the coin lying in front of you is not a fake, which means that the probability for landing on heads and tails is the same (p = ½). Or, you might have a die in front of you and make a hypothesis that it is not loaded, which means that all 6 numbers have the same probability of occurring (p = 1/6). In this chapter you will learn a method to test your hypothesis, however not with absolute certainty as you assume a 5% probability (significance level) with which the null hypothesis is wrongly rejected.

PROGRAMMING CONCEPTS: *Null hypothesis, significance, dispersion, Chi-square test*

A SIGNIFICANTLY FAKE COIN

You begin with the null hypothesis that the coin is not a fake and if you toss it n = 100 times, you get heads a certain number of times *k* and tails *n - k* times.

You repeat the test several times, let's say z = 10,000 times, and the result is a distribution for *k* that you can determine with a simulation. As you expect, it is in a bell-shaped distribution around the average value m = 50 [**more...**].



You now take on the interesting question of in which area +- *s* around the average value a predetermined percentage lays, e.g. 68 % of all tests. You can also determine *s*, called **dispersion**, in the computer simulation by adding up the frequencies to the left and right starting at the average until you reach 6800.

If you mark the area corresponding to 95% of all cases, you obtain approximately double the dispersion.

```
from gpanel import *
import random

n = 100 # size of the test  group
p = 0.5
z = 10000

def showDistribution():
    setColor("blue")
    lineWidth(4)
    for t in range(n + 1):
        line(t, 0, t, h[t])

def showMean():
    global mean
```

```
    sum = 0
    for t in range(n + 1):
        sum += h[t] * t
    mean = int(sum / z + 0.5)
    setColor("red")
    lineWidth(2)
    line(mean, 0, mean, 1000)
    text(mean - 1, -30, str(mean))

def showSpreading(level):
    sum = h[mean]
    for s in range(1, 20):
        sum += h[mean + s] + h[mean - s]
        if sum > z * level:
            break
    setColor("green")
    lineWidth(2)
    line(mean + s, 0, mean + s, 1000)
    text(mean + s - 1, -30, str(mean + s))
    line(mean - s, 0, mean - s, 1000)
    text(mean - s - 1, -30, str(mean - s))

def sim():
    sum = 0
    repeat n:
       w = random.random()
       if w < p:
           sum +=1
    return sum

makeGPanel(-0.1 * n, 1.1 * n, -100, 1100)
title("Coin toss,  distribution of number")
drawGrid(0, n, 0, 1000)
h = [0] * (n + 1)

repeat z:
    k = sim()
    h[k] += 1

showDistribution()
showMean()
showSpreading(0.68)
showSpreading(0.95)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

If you frequently make a test with 100 coins that are not fake, in 68 % of all cases the number of tossed heads lies in the area 50 +-5, and 95% of all cases in the area 50 +-10 [**more...**] .

If you make a test with the coin that is lying in front of you and you get a value for the number of heads that is greater than 60 or smaller than 40 you reject the hypothesis that the coin is not fake, in other words, you say that the coin is fake. In this case, you may be mistaken with a probability of 5% (the significance level). Sometimes you can also concisely say that **the *present coin is significantly fake***.

A SIGNIFICANTLY LOADED DIE

You have a die in front of you and want to test whether it is a fair die, which means that all numbers can occur with the same probability of 1/6. You make the hypothesis: *The die is not loaded*.

Here you will get to know a slightly different method from one that we used with the coin since there are six, not only two, possibilities that can occur on a roll, namely the numbers from 1 to 6. To be on the safe side you will want to roll the die often, let's say around 600 times, and write down the frequencies of the numbers that occur.

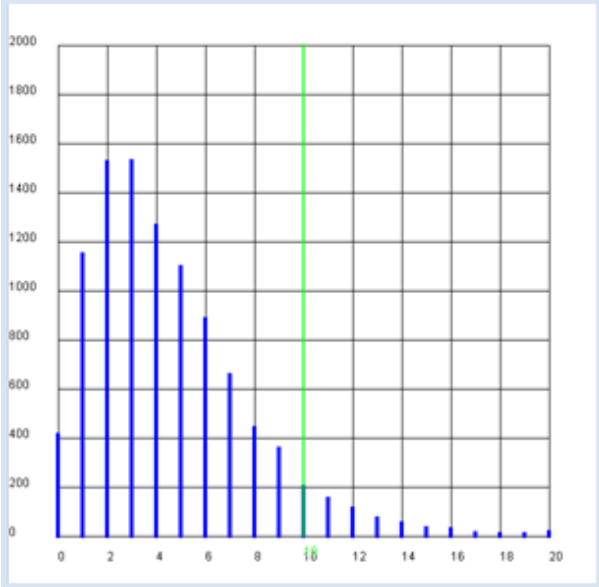
Pip number	Observed frequency (u)	Theoretical frequency (expected value e)
1	112	100
2	128	100
3	97	100
4	103	100
5	88	100
6	72	100
Total	600	600

Observed and theoretical frequencies

In order to introduce a measure for the deviation of the observed from the theoretical occurrences, you need to calculate the relative square deviation for each number $(u - e)^2 / e$ and add up these values. We call the result χ^2 (pronounced "Chi-square").

This raises the interesting question of how χ^2 is distributed, meaning how often the different values of χ^2 occur in many 600-roll attempts. To find this out, perform another computer simulation with 10,000 samples and determine the distribution. For the sake of simplicity, you can round the obtained values to whole numbers [**more...**].

Coincidentally, you again enter a critical value for χ^2, below 95% of all cases. The simulation results in s = 11 [**more...**].




```
from gpanel import *
import random

n = 600 # number of tosses
p = 1 / 6
z = 10000

def showDistribution():
    setColor("blue")
    lineWidth(4)
    for i in range(21):
        line(i, 0, i, h[i])

def showLimit(level):
    sum = 0
    for i in range(21):
        sum += h[i]
        if sum > z * level:
            break
    setColor("green")
    lineWidth(2)
    line(i, 0, i, 2000)
    text(i, -80, str(i))
```

```
    return i

def chisquare(u):
    chisquare = 0
    e = n * p
    for i in range(1, 7):
        chisquare += ((u[i] - e) * (u[i] - e)) / e
    return chisquare

def sim():
    u = [0] * 7
    repeat n:
        t = random.randint(1, 6)
        u[t] += 1
    return chisquare(u)

makeGPanel(-2, 22, -200, 2200)
title("Chi-square simulation  is being carried out. Please wait...")
drawGrid(0, 20, 0, 2000)
h = [0] * 21

repeat z:
    c = int(sim())
    if c < 20:
        h[c] += 1
    else:
        h[20] += 1

title("Chi-square test on  the die")
showDistribution()
s = showLimit(0.95)

# Observed series
u1 = [0, 112, 128, 97, 103, 88, 72]
u2 = [0, 112, 108, 97, 113, 88, 82]
c1 = chisquare(u1)
c2 = chisquare(u2)
print "Die with", u1, "Xi-square:", c1, "loaded?", c1 > s
print "Die with", u2, "Xi-square:", c2, "loaded?", c2 > s
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The computer simulation exposes the following result: in 95% of all cases, χ^2 is less than or equal to the critical value 11. Hence, you have found a method to test if your die is rigged: calculate χ^2 from the observed frequency. If the value is greater than 11, you can say with a 5% probability of being wrong that your null hypothesis of it being a fair die is incorrect, and therefore the die is loaded.

The frequencies of the table above result in $\chi 2 = 18.7$. In other words, the die has a very high probability of being loaded. With another die rolled 600 times you get the frequencies u2 = [112, 108, 97, 113, 88, 82]. Since you obtain $\chi 2 = 8.5$, there is a low probability that the die is loaded.

DIFFERENCES IN HUMAN BEHAVIOR

You can also apply the χ^2 test to a study of the behavior of two groups of people. An interesting question often asked is whether in a particular context the behavior of females and males should be appraised to be statistically different, or whether both sexes behave equally.

You assume that the use of Facebook is studied in a secondary school. A total of 106 girls

(women) and 86 boys (men) were asked whether they have a Facebook account. The survey results are as follows:

	Facebook Yes	Facebook No	Total	% Yes
Females	87	19	106	82.0%
Males	62	24	86	72.1%
Total	149	43	192	77.7%

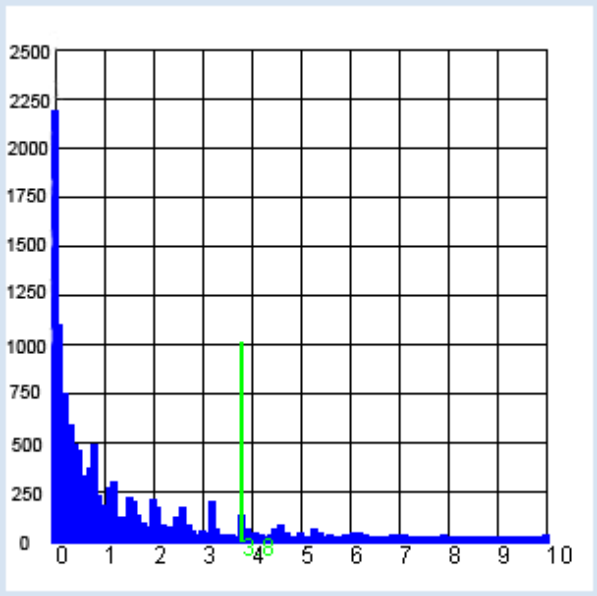
The percentage of people who have a Facebook account is substantially greater among females than it is with males. But it raises the question of whether this higher proportion is statistically significant.

For the simulation, you first determine the probability *p* of having an account from the total number *n* of females and males:

p = (females_yes + males_yes) / n

With this value you simulate the number of females who have an account using random numbers and the total number of females. This results in f0 females with an account and f1 females without one. You do the same for the males, and you will get m0 males with an account and m1 men without one. These numbers form the values u in the calculation of χ^2.

$$\chi^2 = sum\ of\ (u - e)^2 / e$$



You must now still determine the expected value *e* for all four cases. You can assume that p = (f0 + m0) / n is the total probability for a Yes and correspondingly 1 - p is the total probability for a No, so you calculate:

Expected value for females- Yes: ef0 = total number of females * p
Expected value for males- Yes: em0 = total number of males * p
Expected value for females- No: ef1 = total number of females * (1 - p)
Expected value for men- No em1 = total number of males * p

The rest of the program remains largely unchanged from the die test.

```
from gpanel import *
import random

z = 10000
# survey values/polls
females_yes = 87
females_no = 19
males_yes = 62
males_no = 24

def showDistribution():
    setColor("blue")
    lineWidth(4)
    for i in range(101):
        line(i/10, 0, i/10, h[i])

def showLimit(level):
    sum = 0
    for i in range(101):
```

```
        sum += h[i]
        if sum > level * z:
            break
    setColor("green")
    lineWidth(2)
    limit = i / 10
    line(limit, 0, limit, 1000)
    text(limit, -80, str(limit))
    return limit

def chisquare(f0, f1, m0, m1):
    # f: females, m: males, 0:yes, 1:no
    w = (f0 + m0) / n # probability of a yes
    # expected value
    ef0 = (f0 + f1) * w # females-yes
    em0 = (m0 + m1) * w # males-yes
    ef1 = (f0 + f1) * (1 - w) # females-no
    em1 = (m0 + m1) * (1 - w) # males-no
    # add up deviations (u - e)*(u - e) / e
    chisquare = (f0 - ef0) * (f0 - ef0) / ef0 \
              + (m0 - em0) * (m0 - em0) / em0 \
              + (f1 - ef1) * (f1 - ef1) / ef1 \
              + (m1 - em1) * (m1 - em1) / em1
    return chisquare

def sim():
    # simulate females
    f0 = 0 # yes
    f1 = 0 # no
    for i in range(females_all):
        t = random.random()
        if t < p:
            f0 += 1
        else:
            f1 += 1
    # simulate males
    m0 = 0 # yes
    m1 = 1 # no
    for i in range(males_all):
        t = random.random()
        if t < p:
            m0 += 1
        else:
            m1 += 1
    return chisquare(f0, f1, m0, m1)

females_all = females_yes + females_no
males_all = males_yes + males_no
n = females_all + males_all  # all
p = (females_yes + males_yes) / n  # probability of yes for all
print "Facebook yes (all):", round(100 * p, 1), "%"
pf = females_yes / females_all
print "Facebook yes (females):", round(100 * pf, 1), "%"
pm = males_yes / males_all
print "Facebook yes (males:)", round(100 * pm, 1), "%"

makeGPanel(-1, 11, -250, 2750)
title("Chi-square test, use of Facebook")
drawGrid(0, 10, 0, 2500)
h = [0] * 101

repeat z:
    c = int(10 * sim())  # magnification factor of 10
    if c < 100:
        h[c] += 1
    else:
        h[100] += 1

showDistribution()
```

```
s = showLimit(0.95)

c = chisquare(females_yes, females_no, males_yes, males_no)
print "critical value:", s
print "observed:", c,
if c <= s:
   print "- the same behavior"
else:
   print "- not the same behavior"
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The result is astonishing: the χ^2significance limit is 3.8 [**more...**]. The survey values resulted in the smaller value of 2.7. Even though the proportion of females with accounts is essentially higher, it cannot be statistically proven that they differ substantially from the males with respect to Facebook.

EXERCISES

1. You have another idea of how to figure out if a die is loaded. Similar to the 100 coin tosses, you repeatedly simulate a roll of a die 600 times and determine the distribution of the numbers.

   Does the die from the above distribution *u1* equally show as a fake with a 5% significance level? What about the die with the distribution *u2*?

2. A classic roulette table has 37 numbers from 0 to 36 that should occur with equal probability. A clever player wants to detect some irregularities in order to increase their chance of winning. They make notes of the frequency of the numbers that occur in 1,000 games and get:

   u = [20, 26, 20, 22, 20, 27, 18, 28, 21, 36, 20, 28, 25, 19, 22, 25, 33, 25, 28, 25, 32, 29, 22, 32, 28, 31, 26, 25, 32, 32, 25, 20, 25, 44, 40, 24, 45]

   Use a χ^2 test to check the null hypothesis that the roulette is fair.

3. In order to scientifically test a medication, it is prescribed in a blind study to two groups of sick people, where one of the groups receives a placebo. The following values were found after the treatment:

	After treatment-cured	After treatment-sick	%of people cured
Treated with medication	22	13	62.9 %
Treated with placebo	11	17	39.3 %

   The proportion of people cured with medical therapy is much greater than those without. Can we assume that the medication is effective?

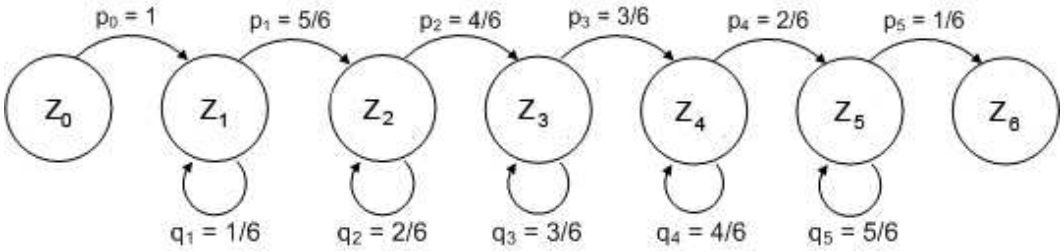
8.4 AVERAGE WAITING TIMES

INTRODUCTION

There are many systems whose temporal behavior can be described as transitions from one state to the next. The transition from a current system state Z_i to the following state Z_k is determined by probabilities p_{ik}. In the example below, you will roll one die and consider the already rolled numbers as a state variables:

Z_0 : no number rolled yet
Z_1 : one number rolled
Z_2 :two (different) numbers rolled
etc.

You can illustrate the transition of the states as in the scheme below (called **Markov chain**):



The probabilities are understood as follows: if you have already rolled *n* numbers, the chances of getting one of these numbers again is n/6 and the chance of getting a number you have not yet rolled is (6 - n)/6. You can attempt to solve the interesting question of how many times you have to roll the die on average to get all six numbers.

PROGRAMMING CONCEPTS: *Markov chain, waiting time, waiting time paradox*

AVERAGE WAITING TIMES

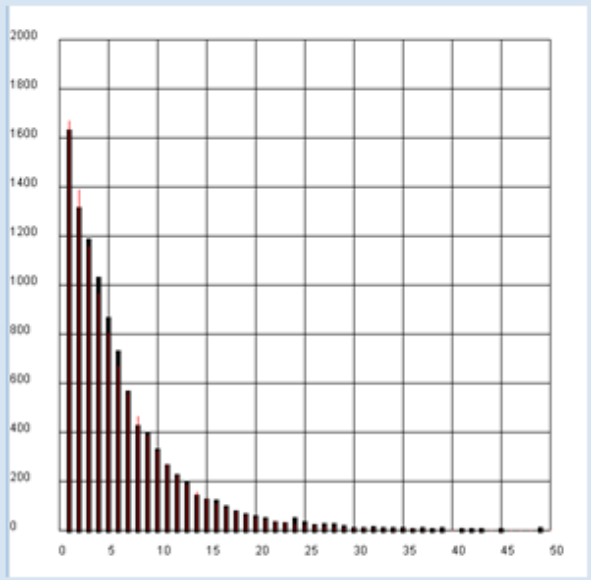
If you roll the die in equal increments of time, you can also ask yourself how long, on average, the process takes. You call this time the **average waiting time.** You can figure it out with the following reflection: the total time to get from Z_0 to Z_6 is the sum of the waiting times for all transitions. But how long are the individual waiting times?

In your first program you can experimentally determine the most important feature of waiting time problems:

If p is the probability of getting from Z_1 to Z_2 the delay time amounts to (in a suitable unit) u = 1/p.

In the simulation you investigate the delay time it takes to roll a certain number, for example a 6. In this case, a simulation attempt does not consist of a single roll of the die. Instead, you roll as many times as it takes to get a 6 in the function *sim()* and you return the number of how many rolls it took. Repeat this experiment 10,000 times and generate the average number of necessary rolls.

At the same time, you display the numbers of attempts in order to get a 6 in a frequency diagram, with k = 1, 2, 3,... (stop at k = 50).




```
from gpanel import *
import random

n = 10000
p = 1/6

def sim():
    k = 1
    r = random.randint(1, 6)
    while r != 6:
        r = random.randint(1, 6)
        k += 1
    return k

makeGPanel(-5, 55, -200, 2200)
drawGrid(0, 50, 0, 2000)
title("Waiting on a 6")
h = [0] * 51
lineWidth(5)
sum = 0
repeat n:
    k = sim()
    sum += k
    if k <= 50:
        h[k] += 1
        line(k, 0, k, h[k])
mean_exp = sum / n

lineWidth(1)
setColor("red")
sum = 0
for k in range(1, 1000):
    pk = (1 - p)**(k - 1) * p
    nk = n * pk
    sum += nk * k
    if k <=50:
        line(k, 0, k, nk)
mean_theory = sum / n
title("Experiment: " + str(mean_exp) + "Theory: " + str(mean_theory))
```

MEMO

The result is intuitively evident: since the probability of rolling a certain number on the die is p= 1/6, you need an average of u = 1 / p = 6 rolls to get this number.

It is also instructive to display the theoretical values of the frequencies as a red line. To do this, you must consider that the following applies to the probability of rolling a 6:

- a 6 in the first roll: $p_1 = p$
- no 6 in the first roll, but in the second roll: $p_2 = (1 - p) * p$
- no 6 in the first or second roll, but in the third roll: $p_3 = (1 - p) * (1 - p) * p$
- a 6 in the k-th roll: $p_k = (1 - p)^{k-1} * p$

To get the theoretical frequencies, multiply these probabilities with the number of trials n [**more...**].

PROGRAMMING INSTEAD OF A LOT OF CALCULATING

In order to solve the task defined above, to calculate the average delay until you have rolled all numbers at least once, you may decide to take the theoretical path. You interpret the process as a Markov chain and add the delay times of each individual transition:

u = 1 + 6/5 + 6/4 + 6/3 + 6/2 + 6 = 14.7

Alternatively, you can write a simple program to determine this number in a simulation. To do this, you always proceed the same way: write a function *sim()* where the computer searches for a single solution using random numbers and then returns the required number of steps as a return value. You then repeat this task many times, let's say 1,000 times, and determine the average value.

It is smart to use a list *z* in *sim()*, where you can insert the rolled numbers that are not already there. Once the list has 6 elements, you have rolled all the numbers of the die.

```
import random

n = 10000

def sim():
    z = []
    i = 0
    while True:
        r = random.randint(1, 6)
        i += 1
        if not r in z:
            z.append(r)
        if len(z) == 6:
            return i

sum = 0
repeat n:
    sum += sim()

print "Mean waiting time:", sum / n
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

In the computer simulation you get the slightly fluctuating value 14.68, which corresponds to the theoretical prediction. The computer can thus also be used to quickly check if a theoretically calculated value can be correct.

However, the theoretical determination of waiting times can already be very complex with simple problems. If you are, for instance, trying to figure out the average waiting time until you have rolled a certain sum of numbers, the problem is extremely easy to solve with a computer simulation

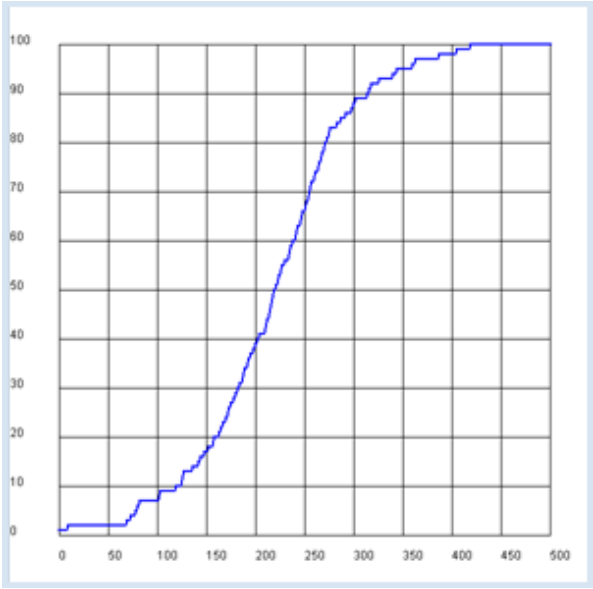
```
import random

n = 10000
s = 7  # rolled sum of the die numbers

def sim():
    i = 0
    total =  0
    while True:
        i += 1
        r = random.randint(1, 6)
        total += r
        if total >= s:
            break
    return i

sum = 0
repeat n:
    sum += sim()

print "Mean waiting time:", sum / n
```

MEMO

You get an average delay time of about 2.52 for rolling the sum 7 with the die. This result is somewhat surprising, given that the expected value for the die numbers is 3.5. Therefore, it could be assumed that you have to roll the die 2x on average to achieve the sum 7. The theoretical calculation, which might take you several hours, results in: 117 577 / 46 656 = 2.5008. Even mathematicians thus use the computer to quickly test theoretical results and to verify speculations.

THE SPREADING OF AN DISEASE

Assume the following story. Even though it is fictional, there are certain parallels to current living communities.
"100 people live on a remote Caribbean island that is cut off from the outside world. An old man is infected by thoughtlessly consuming a migrating bird that had a contagious disease. When a sick individual meets with a healthy individual, they also get sick in a short amount of time. Every hour two people meet by chance."
You want to investigate how the disease spreads using a computer simulation. To do this, you determine the number of people infected in relation to time.



It is smart to model the population with a list with Boolean values, where healthy is coded as *False* and ill as *True*. The advantage of this data structure is that in the function *pair(),* the

interaction that occurs when two people meet can simply be expressed by a logical OR operation:

1. Person previously	2. Person previously	1.& 2. Person afterwards
healthy (False)	healthy (False)	healthy (False)
healthy(False)	ill (True)	ill (True)
ill (True)	healthy (False)	ill (True)
ill (True)	ill (True)	ill (True)

```
from gpanel import *
import random

def pair():
    # Select two distinct inhabitants
    a = random.randint(0, 99)
    b = a
    while b == a:
        b = random.randint(0, 99)
    z[a] = z[a] or z[b]
    z[b] = z[a]

def nbInfected():
    sum = 0
    for i in range(100):
        if z[i]:
            sum += 1
    return sum

makeGPanel(-50, 550, -10, 110)
title("The spread of an illness")
drawGrid(0, 500, 0, 100)
lineWidth(2)
setColor("blue")

z = [False] * 100
tmax =  500
t = 0
a = random.randint(0, 99)
z[a] = True # random infected inhabitant
move(t, 1)

while t <= tmax:
    pair()
    infects = nbInfected()
    t += 1
    draw(t, infects)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

You will find a temporal behavior where the increase is first slow, then rapid, and then slow again. This behavior is certainly feasible qualitatively since the probability is at first low that an ill person encounters a healthy person, since mostly healthy people meet each other. At the end the probability is again low that a remaining healthy person encounters an ill person, since mainly sick people are meeting each other

One interesting question is how long it takes on average for all of the residents to become ill. You can directly solve this question with a computer simulation where you simulate the same population multiple times and count the steps it takes until everyone is ill.

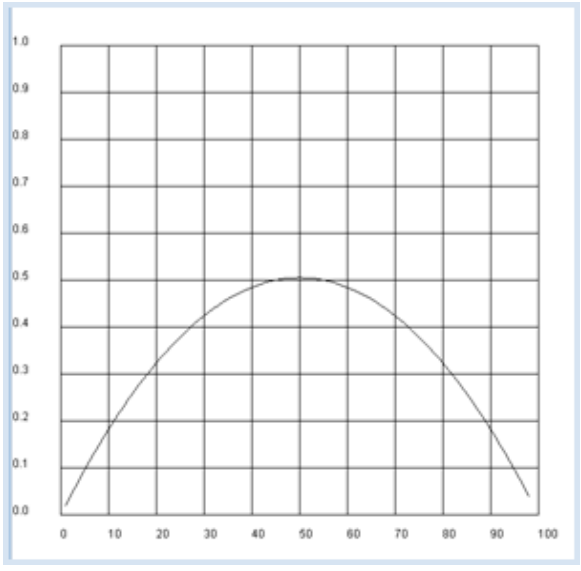
```
import random

n = 1000 # number experiment

def pair():
    # Select two distinct inhabitants
    a = random.randint(0, 99)
    b = a
    while b == a:
        b = random.randint(0, 99)
    z[a] = z[a] or z[b]
    z[b] = z[a]

def nbInfected():
    sum = 0
    for i in range(100):
        if z[i]:
            sum += 1
    return sum

def sim():
    global z
    z = [False] * 100
    t = 0
    a = random.randint(0, 99)
    z[a] = True # random infected inhabitant
    while True:
        pair()
        t += 1
        if nbInfected() == 100:
            return t

sum = 0
for i in range(n):
    u = sim()
    print "Experiment #", i + 1, "Waiting time:", u
    sum += u

print "Mean waiting time:", sum / n
```

You can also illustrate the spread of the disease as a Markov chain. A certain state is characterized by the number of people infected. The time until everyone is ill is the sum of the waiting time for the transitions from *k* ill people to *k+1* ill people, for *k* from 1 to 99. In addition, you need the probability p_k for this transition. It is:

p_k = sum of the probabilities of first choosing an ill person and second a healthy person, and vice versa

$$p_k = \frac{k}{n} * \frac{n-k}{n-1} + \frac{n-k}{n} * \frac{k}{n-1} = 2 * \frac{k*(n-k)}{n*(n-1)}$$

You also illustrate p_k graphically in the program and determine the sum of the reciprocal values.



```
from gpanel import *

n  = 100

def p(k):
    return 2 * k * (n - k) / n / (n - 1)

makeGPanel(-10, 110, -0.1, 1.1)
drawGrid(0, 100, 0, 1.0)

sum = 0
for k in range(1, n - 1):
    if k == 1:
        move(k, p(k))
    else:
        draw(k, p(k))
    sum += 1 / p(k)

title("Time until everyone is ill: " + str(sum))
```

MEMO

Using the theory of Markov chains results in an average waiting time of 463 hours until everyone has the illness, or around 20 days.

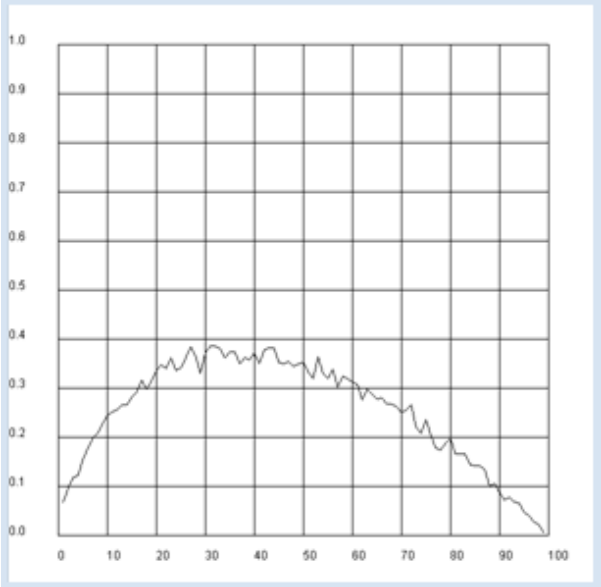
PARTNER SEARCHING WITH A PROGRAM

An interesting matter for practice is the optimal strategy in the selection of a life partner. In this case you start based on the following model assumption: 100 potential partners possess increasing qualification levels. They will be presented to you in a random order and in a "learning phase" you can correctly order them by qualification levels, by relying on your previous life experiences. However, you do not know what the maximum degree of qualification is. With each introduction, you have to decide if you want to accept or reject the partner. What should you do to make sure that you choose the partner with the best qualifications with a high probability?

For the simulation, create a list *t* with the 100 qualification levels from 0 to 99 in any order in the function *sim(x)*. It consists of a random permutation of the numbers 0 to 99, that you can nicely create in *Python* with *shuffle()*.

Subsequently you do the selection process, where you assume a fixed length *x* of the learning phase and determine the index of the chosen partner and their qualification level in the process.

Now you simulate the process with a specific *x* 1,000 times and determine how likely it is that you will get a partner with a maximum qualification level. You eventually depict this probability graphically with respect to the length *x* of the learning phase.




```
import random
```

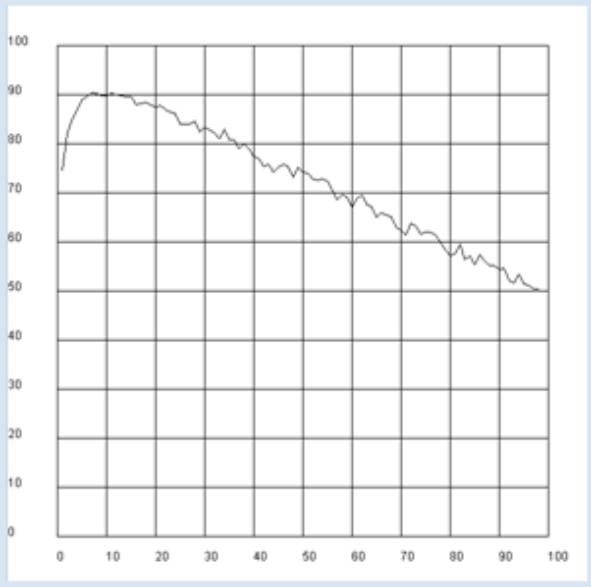
```
from gpanel import *

n = 1000 # Number of simulations
a = 100 # Number of partners

def sim(x):
    # Random permutation [0..99]
    t = [0] * 100
    for i in range(0, 100):
        t[i] = i
    random.shuffle(t)
    best = max(t[0:x])
    for i in range(x, 100):
        if  t[i] > best:
            return [i, t[i]]
    return [99, t[99]]

makeGPanel(-10, 110, -0.1, 1.1)
title("The probability of finding the best partner from  100")
drawGrid(0, 100, 0, 1.0)

for x in range(1, 100):
    sum = 0
    repeat n:
        z = sim(x)
        if z[1] == 99:  # best score
            sum += 1
    p = sum / n
    if x == 1:
        move(x, p)
    else:
        draw(x, p)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Apparently you are the likeliest to find the partner with the best qualifications after a learning phase of around 37 [**more...**].

However, you can also optimize the sampling method following a different criterion where you are not looking for the best partner, but rather one with a qualification level as high as possible. To do this, you examine with a similar simulation the average qualification of the selected partner for a given length *x* of the learning phase.



```
import random
from gpanel import *

n = 1000 # Number of simulations

def sim(x):
    # Random permutation [0..99]
    t = [0] * 100
    for i in range(0, 100):
        t[i] = i
    random.shuffle(t)
    best = max(t[0:x])
    for i in range(x, 100):
        if  t[i] > best:
            return [i, t[i]]
    return [99, t[99]]

makeGPanel(-10, 110, -10, 110)
title("Mean qualification after waiting for a  partner")
drawGrid(0, 100, 0, 100)

for x in range(1, 99):
    sum = 0
    repeat n:
        u = sim(x)
        sum += u[1]
    y = sum / n
    if x == 1:
       move(x, y)
    else:
       draw(x, y)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

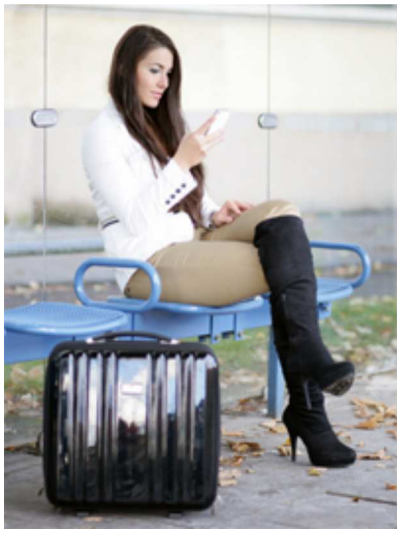
MEMO

This looks entirely different: following the criterion of the best possible average qualification, you should already make your decision for the next better rated partner after a learning phase of about 10 .

You can also perform the simulation for a more realistic number of partners and notice that the optimal learning phase is quite short [**more...**]

WAITING TIME PARADOX

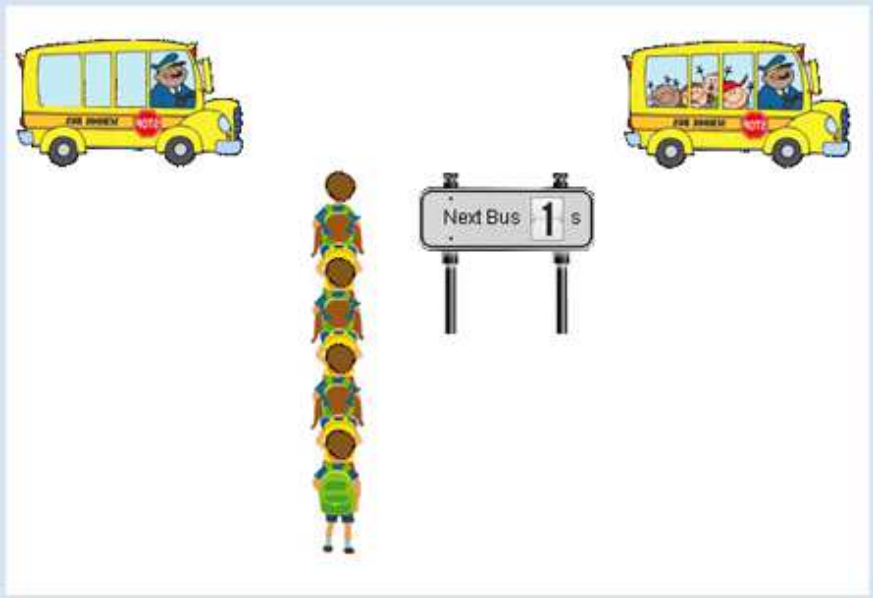
Waiting at a stop for any form of public transportation (bus, tram, train, etc.) is part of everyday life. Here we will consider how long the average wait time is if you go to the stop at a completely random time (in other words, you do not know the schedule). We first assume that the buses make a stop at this station precisely every 6 minutes.



It is clear that sometimes you might have to wait just a short time, and other times almost the maximum of 6 minutes. This makes the waiting period 3 minutes on average. However, how is it when the buses do not run at exact intervals, but rather run, for example, with a uniformly distributed probability in a range from 2 to 10 minutes?

Since they still come by every 6 minutes on average in this case, it is possible that you assume that the waiting time of 3 minutes stays the same. The surprising and therefore paradoxical answer is that the average waiting time is greater than 3 minutes in this situation.

In an animated simulation you should find out how long the waiting time is under the assumption that the buses are uniformly distributed to arrive one after another between every 2 to 10 minutes (in the simulation these are seconds). You can use the game library *JGameGrid* for this since you can easily model objects such as buses and passengers as sprite objects.



The program code requires some explanation:

Since we are obviously dealing with bus and passenger objects, they are modeled by the classes *Bus* and *Passenger*. The buses are created in an infinite loop at the end of the main part of the program according to the statistical requirements. When the graphics window is closed the infinite loop breaks due to *isDisposed() = False* and the program ends.

The passengers must be periodically generated and displayed in the queue. The best way to do this, is to write a class *PassengerFactory* that is derived from *Actor*. Even though this does not have a sprite image, its *act()* can be used to generate passengers and to insert them into the GameGrid. You can select the period at which the objects are generated using the cycle counter *nbCycles* (the simulation cycle is set to 50 ms).

You move the bus forward in the *act()* of the class *Bus* and check if it has arrived at the stop with the x-coordinate. When it arrives, you call the method *board()* of the class *PassengerFactory* whereby the waiting passengers are removed from the queue. Simultaneously change the sprite image of the buses with *show(1)* and show the new waiting time for the next bus on the scoreboard. Use the Boolean variable *isBoarded* so that these actions are only called once.

The scoreboard as an instance of the class *InformationPanel* is an additional gadget to show the time it will take until the next bus arrives. The display will again be changed in the method *act()* by selecting one of the 10 sprite images (*digit_0.png* to *digit_9.png)* using *show()*.

```
from gamegrid import *
import random
import time

min = 2
max = 10

def random_t():
    return min + (max - min) * random.random()

# ---------------- class PassengerFactory ----------
class PassengerFactory(Actor):
```

```
    def __init__(self):
        self.nbPassenger = 0

    def board(self):
        for passenger in getActors(Passenger):
            passenger.removeSelf()
            passenger.board()
        self.nbPassenger = 0

    def act(self):
        if self.nbCycles % 10 == 0:
            passenger = Passenger(random.randint(0, 1))
            addActor(passenger, Location(400, 120 + 27 * self.nbPassenger))
            self.nbPassenger += 1

# ---------------- class Passenger -----------------
class Passenger(Actor):
    totalTime = 0
    totalNumber = 0

    def __init__(self, i):
        Actor.__init__(self, "sprites/pupil_" + str(i) + ".png")
        self.createTime = time.clock()

    def board(self):
        self.waitTime = time.clock() - self.createTime
        Passenger.totalTime += self.waitTime
        Passenger.totalNumber += 1
        mean = Passenger.totalTime / Passenger.totalNumber
        setStatusText("Mean waiting time: " + str(round(mean, 2)) + " s")

# ---------------- class Car -----------------------
class Bus(Actor):
    def __init__(self, lag):
        Actor.__init__(self, "sprites/car1.gif")
        self.lag = lag
        self.isBoarded = False

    def act(self):
        self.move()
        if self.getX() > 320 and not self.isBoarded:
            passengerFactory.board()
            self.isBoarded = True
            infoPanel.setWaitingTime(self.lag)
        if self.getX() > 1650:
            self.removeSelf()

# ---------------- class InformationPanel ----------
class InformationPanel(Actor):
    def __init__(self, waitingTime):
        Actor.__init__(self, "sprites/digit.png", 10)
        self.waitingTime = waitingTime

    def setWaitingTime(self, waitingTime):
        self.waitingTime = waitingTime

    def act(self):
        self.show(int(self.waitingTime + 0.5))
        if self.waitingTime > 0:
            self.waitingTime -= 0.1

periodic = askYesNo("Departures every 6 s?")
makeGameGrid(800, 600, 1, None, None, False)
addStatusBar(20)
setStatusText("Acquiring data...")
setBgColor(Color.white)
setSimulationPeriod(50)
show()
doRun()
```

```
if periodic:
    setTitle("Warting Time Paradoxon - Departure every 6 s")
else:
    setTitle("Waiting Time Paradoxon - Departure between 2 s and 10 s")

passengerFactory = PassengerFactory()
addActor(passengerFactory, Location(0, 0))

addActor(Actor("sprites/panel.png"), Location(500, 120))
addActor(TextActor("Next Bus"), Location(460, 110))
addActor(TextActor("s"), Location(540, 110))
infoPanel = InformationPanel(4)
infoPanel.setSlowDown(2)
addActor(infoPanel, Location(525, 110))

while not isDisposed():
    if periodic:
        lag = 6
    else:
        lag = random_t()
    bus = Bus(lag)
    addActor(bus, Location(-100, 40))
    a = time.clock()
    while time.clock() - a < lag and not isDisposed():
        delay(10)
```

MEMO

The simulation shows that the average waiting time is around 3.5 minutes, so in other words clearly longer than the previously assumed 3 minutes. You can explain this change as follows: If the buses arrive equally distributed, once with a gap of 2 minutes and once with a gap of 10 minutes, it is much more likely that you will get to the bus stop during the waiting interval from 2 to 10 minutes, rather than 0 to 2 minutes. Therefore, you will certainly end up waiting longer than 3 minutes.

EXERCISES

1. You get either a ten, twenty, or fifty cent coin every day, each with the probability of 1/3. How many days go by, on average, until you can buy a book that costs ten dollars?

2. A person who is lost starts in the middle of the window and moves 10 pixels with each step in a random direction (random walk). What is the average delay time *u* until they have for the first time moved farther than the distance *r* from the start? Simulate the movement with a (hidden) turtle with the values r = 100, 200, 300. What do you think is the relationship between *r* and *u*?

3. Modify the program for the waiting time paradox so that the buses arrive either 2 or 10 seconds apart with the same probability of ½ and determine the average waiting time.

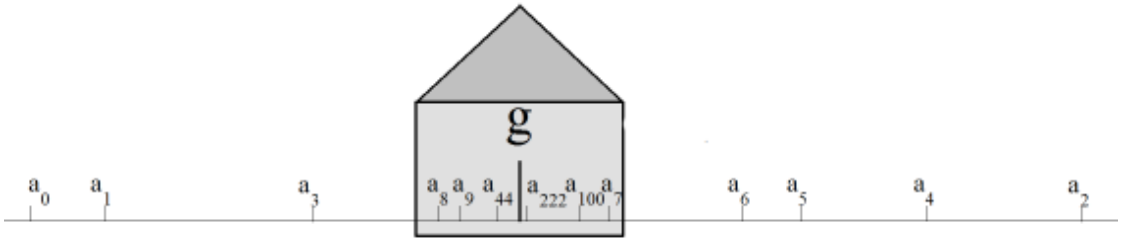
8.5 SEQUENCES, CONVERGENCE

INTRODUCTION

Sequences of numbers have fascinated people for a long time. Already in ancient times around 200 B.C. Archimedes had approximated the number of Pi through a sequence of numbers that he found by calculating the perimeters of regular polygons with increasingly many vertices that fit into a circle. It was already clear to him then that the circle could be regarded as a border-line case of a regular polygon with infinitely many vertices. Therefore, he found that the perimeters of the polygons had to **strive** towards the circumference of the circle.

A sequence of numbers consists of the numbers a_0, a_1, a_2, etc., thus of the terms an with index n = 0, 1, 2, etc. (sometimes it begins with n = 1). A formation rule clearly determines how big each number is. If a_k is defined for any natural number, however large it may be, we speak of an infinite sequence of numbers. The principle can be specified as an explicit expression of *n*. However, a term may also be calculated from previous terms of the series (recursive rule). In this case, the start values must also be known.

A convergent sequence has a very descriptive property: there is a unique number *g* called a **limit** that the terms gradually approach, which means that you can choose any arbitrarily small neighborhood interval of *g* so that none of the following terms fall outside of the chosen interval starting at a definite *n*. You can imagine the neighborhood as a house around the limit: before that, the sequence may leap around wildly, however after a certain *n* all terms of the sequence end up in the house no matter how small it may be.



Number sequences can be studied experimentally using the computer. In order to illustrate them, you use various graphical representations: You can, for instance, similar to the above illustration, draw all terms as points or lines and analyze whether there is a **limit point**. However, you can also investigate how the sequence behaves **for large values of n** by representing it as a two dimensional graph where the *n* are on the x-axis and the *an* on the y-axis.

PROGRAMMING CONCEPTS: *Limit point, convergence, tree diagram, chaos*

THE HUNTER AND HIS DOG

A hunter walks with his dog with the velocity u = 1 m/s to their hunting lodge d = 1000 m away. However, since the hunter walks too slowly for the dog, the dog proceeds as follows: It runs alone at its own velocity u = 20 m/s to the hunting lodge, turns around, and then runs back to its owner. As soon as it reaches its owner, it turns around again and runs back to the hunting lodge, and then continues this behavior.

You would like to simulate this procedure with a program. With every press of the button in your program, you draw the next meeting point of the hunter and the dog and write out the position next to it. The successive values form a sequence of numbers whose behavior you want to study. Since the same amount of time passes for the hunter and the dog between each of their meetings, the increase in position of the hunter can be described as follows:

$$\frac{da}{u} = \frac{2 * (d - a) - da}{v}$$

From this, you can deduce the following relationship with little knowledge of Algebra:

$$da = c * (d - x) \qquad \text{mit} \qquad c = \frac{2 * u}{u + v}$$

As you might have expected, the numbers a_n pile up against the limit number 1000.

```
from gpanel import *

u = 1 # m/s
v = 20 # m/s
d = 1000  # m
a = 0     # hunter
h = 0     # dog
c = 2 * u / (u + v)
it = 0

makeGPanel(-50, 50, -100, 1100)
title("Hunter-Dog problem")
line(0, 0, 0, 1000)
line(-5, 1000, 5, 1000)
line(-5, 1050, 5, 1050)
line(-5, 1000, -5, 1050)
line(5, 1000, 5, 1050)

while not isDisposed():
    move(0, a)
    fillCircle(1)
    text(5, a, str(int(a)))
    getKeyWait()
    da = c * (d - a)
    dh = 2 * (d  - a) - da
    h += dh
    a += da
    it += 1
    title("it = " + str(it) + "; hunter = " + str(a) +
          " m; dog = " + str(h) + " m")
```

MEMO

One says that the number sequence a_n **converges** and that its limit value is 1000.

Think about why this problem is of a theoretical nature. It corresponds to the ancient anecdote where Achilles was invited to race with a turtle. Since he was ten times faster than the turtle, the turtle would have gotten a head start of 10 meters. Achilles refused to compete because in

his opinion, he had no chance to catch up with the turtle. So, he argued: In the time that he needed for the first 10 meters, the turtle would have already advanced 1 meter. In the time that he needed for this one meter, the turtle would already be 10 cm further. In the time that he then needed for these 10 cm, the turtle would already be another 1 cm further, and so forth. What do you think about this?

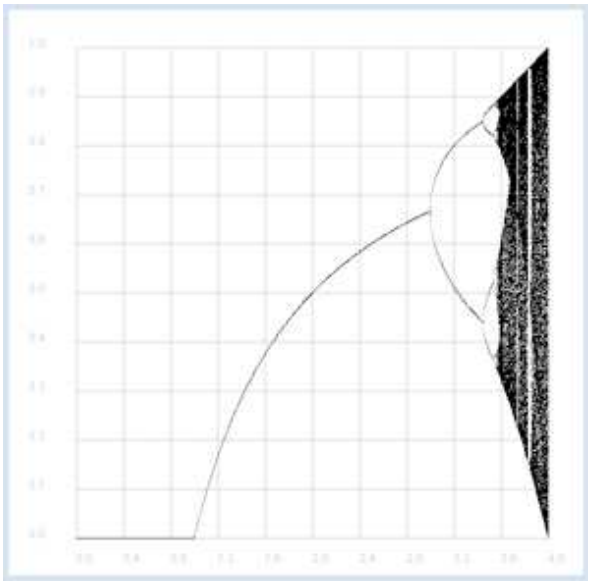
BIFURCATION DIAGRAM (FEIGENBAUM DIAGRAM)

You got to know the logistic growth in the context of population dynamics. The population size *xnew* in the next generation is calculated from its current size *x* from a quadratic relationship. The relationship is simplified in the following (parameter *r* can be arbitrarily chosen) :

$$xnew = r * x * (1 - x)$$

$$a_{n+1} = r * a_n * (1 - a_n)$$

You wonder whether the resulting recursively defined sequence with $a_0 = 0.5$ converges and what the limit value is in this case.



You examine the behavior with an extremely simple program where you plot the first 1,000 terms of the sequence as points for 1,000 equidistant values of *r* in an area from 0 to 4.

With a fixed *r,* you should always begin with the same starting value $a_0 = 0.5$ and draw the terms only from n = 500, since you are only interested in finding out if the sequence is convergent or divergent.

```
from gpanel import *

def f(x, r):
     return r * x * (1 - x)

makeGPanel(-0.6, 4.4, -0.1, 1.1)
title("Tree Diagram")
drawGrid(0, 4.0, 0, 1.0, "gray")
for z in range(1001):
    r = 4 * z / 1000
    a = 0.5
    for i in range(1001):
        a = f(a, r)
        if i > 500:
            point(r, a)
```

MEMO

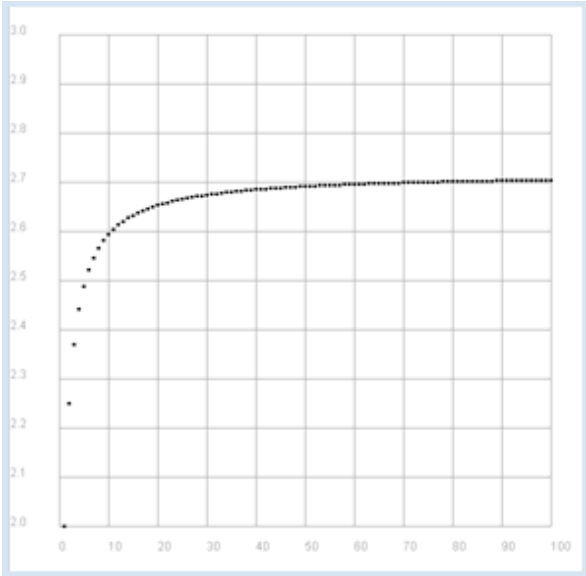
In the experiment, you detect the limit points of the sequence for a certain *r*. Based on a computer simulation, you can establish the following assumptions: For r < 1 there is a limit point at 0 and so the sequence converges to 0. The sequence likewise converges in the area between 1 and r. There are initially two and later more limit points for an even larger *r*, but the sequence no longer converges. For even larger values of *r* the sequence chaotically jumps back and forth.

THE EULER NUMBER

One of the most famous sequences are the numbers defined by the formation rule:

$$a_n = (1 + \frac{1}{n})^n \quad mit\ n = 1, 2, 3, \ldots$$

It is not clear what this sequence does with an increasing n, since on the one hand 1 + 1/n increasingly approaches the number 1, and on the other hand, this number is exponentiated with an increasing exponent. You can try to solve this mystery with a simple computer experiment.




```
from gpanel import *

def a(n):
     return (1 + 1/n)**n

makeGPanel(-10, 110, 1.9, 3.1)
title("Euler Number")
drawGrid(0, 100, 2.0, 3.0, "gray")
for n in range(1, 101):
    move(n, a(n))
    fillCircle(0.5)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

$a_n = (1 + \frac{1}{n})^n$ The sequence converges to a number in the order of 2.7.

This is a case of the **Euler number e**, arguably one of the most famous numbers ever.

QUICK CONVERGING SEQUENCES FOR THE CALCULATION OF PI

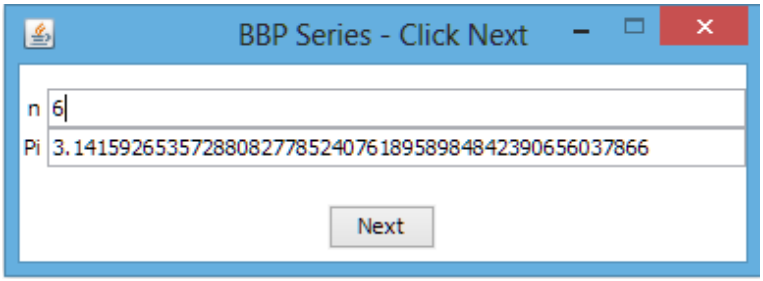
The calculation of π for as many digits as possible poses a challenge since ancient times. A sum formula was discovered only in 1995 by the mathematicians Bailey, Borwein and Plouffe called the BBP formula. They proved that you can get exactly π as the limit value of a sequence whose *n*-th term is the sum from k = 0 to k = n of:

$$\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

Your program uses the *Python* class **decimal** which provides the decimal numbers with a high level of accuracy. The constructor creates such a number from an integer or a float where common mathematical operation signs can be used directly.

You can set the accuray with *getcontext().prec*. This roughly corresponds to the number of decimal places used.

Each time a button is pressed, your program calculates the next term of the sequence and represents the value in a EntryDialog.




```
from entrydialog import *
from decimal import *
getcontext().prec = 50

def a(k):
    return 1/16**Decimal(k) * (4 / (8 * Decimal(k) + 1) - 2
    / (8 * Decimal(k) + 4) - 1
    / (8 * Decimal(k) + 5) - 1 / (8 * Decimal(k) + 6))

inp = IntEntry("n", 0)
out = StringEntry("Pi")
pane0 = EntryPane(inp, out)
btn = ButtonEntry("Next")
pane1 = EntryPane(btn)
dlg = EntryDialog(pane0, pane1)
dlg.setTitle("BBP Series - Click Next")
n = 0
s = a(0)
out.setValue(str(s))
while not dlg.isDisposed():
    if btn.isTouched():
        n = inp.getValue()
        if n == None:
            out.setValue("Illegal entry")
        else:
            n += 1
            s += a(n)
            inp.setValue(n)
            out.setValue(str(s))
```

MEMO

At just 40 iterations the displayed number for π no longer changes.

The program ends when you close the display window, since *isDisposed()* is *True*.

EXERCISES

1. The Fibonacci sequence is defined such that a term is equal to the sum of its two predecessors, with the first and second terms being 1. Calculate the first 30 terms of the sequence and display them on a x-y graph.

2. The Fibonacci sequence diverges, whereas the sequence of quotients of two consecutive terms converges. Similar to exercise 1, display this sequence of quotients graphically and determine the approximate value of the limit.

3. Consider the following sequence with the start value $a_0 = 1$:

$$a_{n+1} = \frac{1}{2} * \left(a_n + \frac{2}{a_n} \right)^n$$

Draw the terms as points on a number line and write their value in an output window. As you can see, the sequence converges. You can calculate the limit value *x* somewhat generously as follows: For large *n* subsequent terms may not be easily distinguishable, so in the border line case we get:

$$x = \frac{1}{2} * (x + \frac{2}{x}) \quad \text{und aufgelöst} \quad x = \sqrt{2}$$

Formulate this result as a guide on how you can approximately determine the square root of 2 using the 4 basic arithmetic operations. How does the algorithm need to be changed for the determination of the square root of any given number *z*?

ADDITIONAL MATERIAL

ITERATIVE SOLUTION OF AN EQUATION

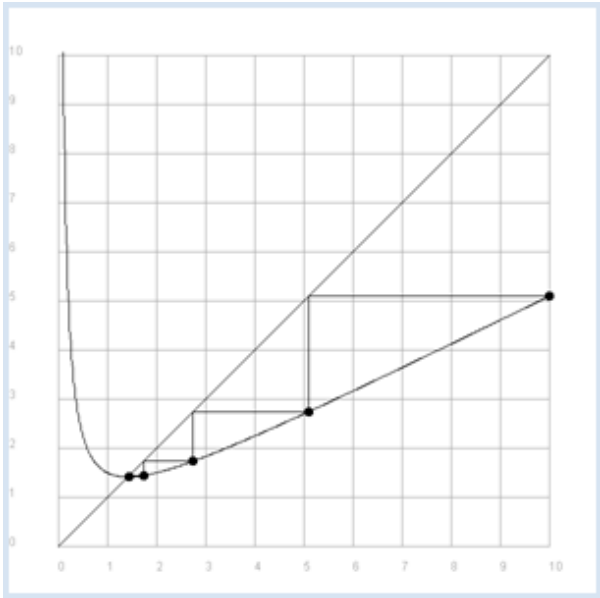
Wie du in Aufgabe 3 gesehen hast, ist $x = \sqrt{2}$ die Lösung der Gleichung

$$x = f(x) \quad mit \quad f(x) = \frac{1}{2} * (x + \frac{2}{x})$$

It is illustrative to display the procedure of solving this equation graphically. To do this, draw both the function graph y = f(x) and the angle bisector y = x in the same coordinate system. The solution is at the intersection of the two curves.

The iterative solution corresponds to the successive pass of a point on the function graph horizontally towards the bisecting angle and vertically down towards the nearest point of the function graph.

.



```
from gpanel import *

def f(x):
    return 1 / 2 * (x + 2 / x)

makeGPanel(-1, 11, -1, 11)
title("Iterative square root begins at x = 10. Press a  key...")
drawGrid(0, 10, 0, 10, "gray")

for i in range(10, 1001):
    x = 10 / 1000 * i
    if i == 10:
        move(x, f(x))
    else:
        draw(x, f(x))

line(0, 0, 10, 10)

x = 10
move(x, f(x))
fillCircle(0.1)
it = 0
```

```
while not isDisposed():
    getKeyWait()
    it += 1
    xnew = f(x)
    line(x, f(x), xnew, f(x))
    line(xnew, f(x), xnew, f(xnew))
    x = xnew
    move(x, f(x))
    fillCircle(0.1)
    title("Iteration " + str(it) + ": x = " + str(x))
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

You can clearly see that with each press of the key the points move quickly towards the point of intersection. Just a few iterations results in a solution with 10 digits of accuracy.

8.6 CORRELATION, REGRESSION

INTRODUCTION

In both the natural sciences and in daily life, the collection of data in the form of measurements plays an important role. However, you do not always need to use a measuring instrument. The data can also consist in poll values, for example, in connection with a statistical investigation. After any data acquisition, it is most important to **interpret** the measured data. You may look for qualitative statements where the measured values rise or fall over time, or you might want experimentally verify a law of nature.

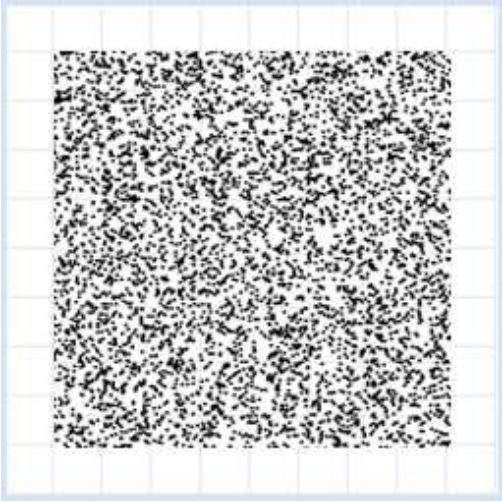
A major problem is that measurements are almost always subject to fluctuations and are not exactly reproducible. Despite these "measurement errors" people would still like to make scientifically correct statements. Measurement errors do not always emerge from flawed measuring instruments, but they may also lie in the nature of the experiment. For instance, the "measurement" of the numbers on a rolled die basically results in dispersed values between 1 and 6. Therefore, with any type of measurement, statistical considerations play a central role.

Often two variables *x* and *y* are measured together in a measurement series and questions arise as to whether these are related and if they are subject to regularity. Plotting the (x, y) values as measuring points in a coordinate system is called **data visualization**. You can almost always recognize whether the data are dependent of each other by simply looking at the **distribution of the measured values**.

PROGRAMMING CONCEPTS: *Data visualization, measured value distribution, cloud diagram, noise, covariance, correlation coefficient, regression, best fit*

VISUALIZING INDEPENDENT AND DEPENDENT DATA

You can easily simulate independent data in a x-y diagram by using uniformly distributed random numbers for x and y. Although it is not necessary in this case, you copy the measured values into the data lists *xval* and *yval* and only then display them as data points.



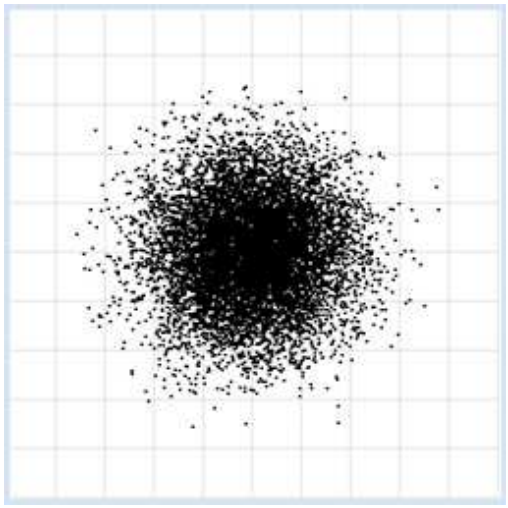
```
import random
from gpanel import *

z = 10000

makeGPanel(-1, 11, -1, 11)
title("Uuniformly distributed value pairs")
drawGrid(10, 10, "gray")

xval = [0] * z
```

```
yval = [0] * z

for i in range(z):
    xval[i] = 10 * random.random()
    yval[i] = 10 * random.random()
    move(xval[i], yval[i])
    fillCircle(0.03)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

You get galaxy-like graphics if the x and y values are still independent of each other, but normally distributed around an average value. In this example, 5 is used as the average and 1 is used as the dispersion. This is also called a **scatter plot** or a **cloud diagram**.

You can easily generate normally distributed random numbers by using *random.gauss(average value, dispersion)*.

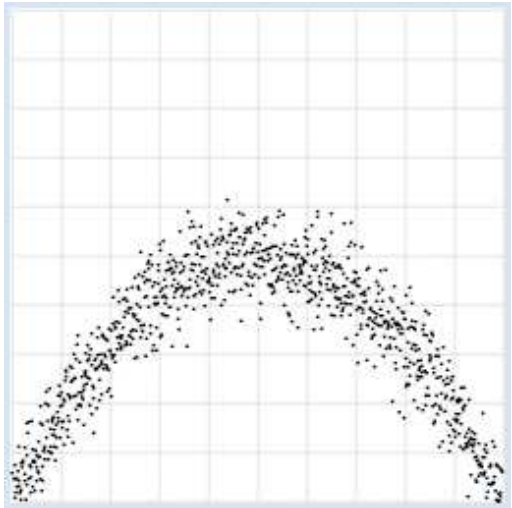


```
import random
from gpanel import *

z = 10000

makeGPanel(-1, 11, -1, 11)
title("Normally distributed value pairs")
drawGrid(10, 10, "gray")

xval = [0] * z
yval = [0] * z

for i in range(z):
    xval[i] = random.gauss(5, 1)
    yval[i] = random.gauss(5, 1)
    move(xval[i], yval[i])
    fillCircle(0.03)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

You can also simulate dependencies between x and y values by assuming that x increases in equidistant steps in a certain range, and y is a function of x that is, however, subject to statistical fluctuations. In physics, such fluctuations are often called **noise**.

You draw the cloud diagram for a parabola y = -x * (0.2 * x - 2) and normally distributed noise in the area x = 0..10 with an increment size of 0.01.



```
import random
from gpanel import *
import math

z = 1000
a = 0.2
b = 2

def f(x):
    y = -x * (a * x - b)
    return y

makeGPanel(-1, 11, -1, 11)
title("y = -x * (0.2 * x - 2) with normally distributed noise")
drawGrid(0, 10, 0, 10, "gray")

xval = [0] * z
yval = [0] * z

for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, 0.5)
    move(xval[i], yval[i])
    fillCircle(0.03)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

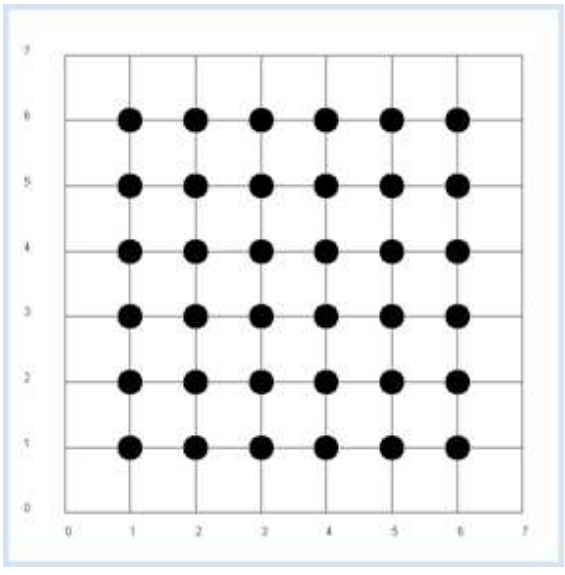
MEMO

You can recognize interdependencies of measured variables immediately through the use of data visualization [**more...**].

With uniformly distributed random numbers in an interval [a, b], the numbers appear with the same frequency in each subinterval of equal length. Normally distributed numbers have a bell-shape distribution, where 68% of the numbers lie inside the interval (average - dispersion) and (average + dispersion).

COVARIANCE AS A MEASURE FOR DEPENDENCY

Here the goal is to not only make interdependencies visible in a diagram, but also to express them by a number. As often, you start from a concrete example and consider the double rolling of a die, where x is the number you get from the first roll and y is the number from the second roll. You conduct the rolling experiment many times and draw the value pairs as a cloud diagram. Since both measurement values of x and y are independent, you obtain a regular point cloud. If you calculate the expected value of x and y, it results in the generally known 3.5.



Since x and y are independent, the probability p_{xy}, to get the pair x, y from a double roll is p_{xy} =

$p_x * p_y$.

It is an obvious assumption that in the general case the following **product rule** applies.

If x and y are independent, the expected value of x * y equals the product of the expected values of x and y [**more...**] .

This assumption is confirmed in the simulation.

```
from random import randint
from gpanel import *

z = 10000 # number of double rolls

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

makeGPanel(-1, 8, -1, 8)
title("Double rolls. Independent random variables")
addStatusBar(30)
drawGrid(0, 7, 0, 7, 7, 7, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = randint(1, 6)
    b = randint(1, 6)
    xval[i] = a
    yval[i] = b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym))
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

It is advisable to use a status bar where the results can be written out. The function *dec2()* rounds the value to 2 digits and returns it as a string.

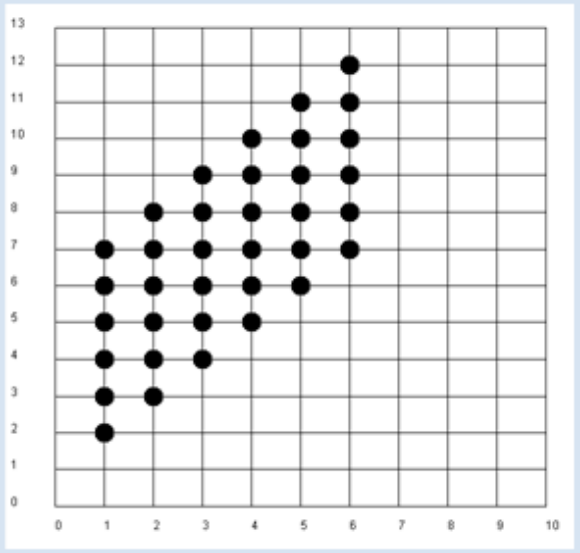
The values are of course subject to a statistical fluctuation.

In the next simulation you will no longer analyze the rolled pair of numbers, but rather the numbers x of the first die and the sum y of the two rolled numbers. It is now evident that y is dependent on x, for example if x = 1, the probability of y = 4 is not the same as when x = 2.

Your simulation confirms that the product rule indeed no longer applies if x and y are interdependent. It is therefore reasonable to introduce the deviation from the product rule, the difference

$$c = E(x*y) - E(x)*E(y)$$

called **covariance,** as a measure of the dependency between x and y. Simultaneously, you see in your program that the covariance can also be calculated as the sum of all square deviations from the average.




```
from random import randint
from gpanel import *

z = 10000 # number of double rolls

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

makeGPanel(-1, 11, -2, 14)
title("Double rolls. Independent random variables")
addStatusBar(30)
drawGrid(0, 10, 0, 13, 10, 13, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = randint(1, 6)
    b = randint(1, 6)
    xval[i] = a
    yval[i] = a + b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
c = xym - xm * ym
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym) + \
```

```
          ",  c = " + dec2(c) + \
          ",  covariance = " + dec2(covariance(xval, yval)))
```

MEMO

You get a value of about 2.9 for the covariance in the simulation. The covariance is therefore well qualified as a measure for the dependence of random variables.

THE CORRELATION COEFFICIENT

The just introduced covariance also has a disadvantage. Depending on how the measured values x and y are scaled, different values occur, even if the values are apparently equally dependent on each other. This is easy to see. If you take, for example, two dice that have sides with numbers going from 10 to 60 instead of from 1 to 6, the covariance changes greatly even though the dependence is the same. This is why you introduce a **normalized covariance**, called a **correlation coefficient**, by dividing the covariance by the dispersion of the x and y values

$$correlation\ coefficient(x, y) = \frac{covariance(x, y)}{dispersion(x) * dispersion(y)}$$

The correlation coefficient is always in the range from -1 to 1. A value close to 0 corresponds to a small dependence and a value close to 1 represents a large dependence, whereas increasing values of x correspond to increasing values of y, and a value close to -1 corresponds to increasing values of x and decreasing values of y.

You again use the double roll in the program and analyze the dependence of the sum of the numbers from the number of the first die.

```
from random import randint
from gpanel import *
import math

z = 10000 # number of double rolls
k = 10 # scalefactor

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx
```

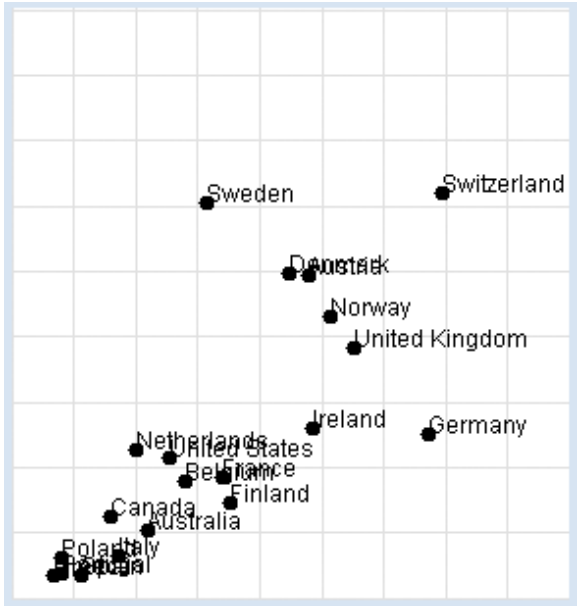
```
def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-1 * k, 11 * k, -2 * k, 14 * k)
title("Double rolls. Independent random variables.")
addStatusBar(30)
drawGrid(0, 10 * k, 0, 13 * k, 10, 13, "gray")

xval =  [0] * z
yval =  [0] * z
xyval = [0] * z

for i in range(z):
    a = k * randint(1, 6)
    b = k * randint(1, 6)
    xval[i] = a
    yval[i] = a + b
    xyval[i] = xval[i] * yval[i]
    move(xval[i], yval[i])
    fillCircle(0.2 * k)

xm = mean(xval)
ym = mean(yval)
xym = mean(xyval)
c = xym - xm * ym
setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  E(x, y) = " + dec2(xym) + \
          ",  covariance = " + dec2(covariance(xval, yval)) + \
          ",  correlation = " + dec2(correlation(xval, yval)))
```

MEMO

You can change the scaling factor *k* and the correlation will stay around 0.71, as opposed to the covariance which greatly changes. Instead of always calling it the correlation coefficient, you can simply call it **correlation**.

MEDICAL RESEARCH PUBLICATION

With the knowledge you have gained, you are already capable of understanding and evaluating a scientific publication, published in the year 2012 in the prestigious journal "New England Journal of Medicine" [**more...**]. The connection between consumers of chocolate and the number of Nobel Prize winners is investigated in different industrialized countries, or in other words, the question is considered whether there is a correlation between eating chocolate and intelligence. In the article the author uses the following data sources which you can also find on the Internet



Nobel Prizes:
http://en.wikipedia.org/wiki/List_of_countries_by_Nobel_laureates_per_capita

Chocolate consumption:
http://www.chocosuisse.ch/web/chocosuisse/en/documentation/facts_figures.html
http://www.theobroma-cacao.de/wissen/wirtschaft/international/konsum

You want to reconstruct the investigation yourself. In your program, you use a list *data* with sub-lists of three elements: the name of the country, the amount of chocolate consumption in kg per year and per inhabitant, and the number of Nobel Prize winners per 10 million inhabitants. You display the data graphically and determine the correlation coefficient.

```
import random
from gpanel import *
import math

data = [["Australia", 4.8, 5.141],
        ["Austria", 8.7, 24.720],
        ["Belgium", 5.7, 9.005],
        ["Canada", 3.9, 6.253],
        ["Denmark", 8.2, 24.915],
        ["Finland", 6.8, 7.371],
        ["France", 6.6, 9.177],
        ["Germany", 11.6, 12.572],
        ["Greece", 2.5, 1.797],
        ["Italy", 4.1, 3.279],
        ["Ireland", 8.8, 12.967],
        ["Netherlands", 4.5, 11.337],
        ["Norway", 9.2, 21.614],
        ["Poland", 2.7, 3.140],
        ["Portugal", 2.7, 1.885],
        ["Spain", 3.2, 1.705],
        ["Sweden", 6.2, 30.300],
        ["Switzerland", 11.9, 30.949],
        ["United Kingdom", 9.8, 19.165],
        ["United States", 5.3, 10.811]]

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-2, 17, -5, 55)
drawGrid(0, 15, 0, 50, "lightgray")
```

```
xval = []
yval = []

for country in data:
    d = country[1]
    v = country[2]
    xval.append(d)
    yval.append(v)
    move(d, v)
    fillCircle(0.2)
    text(country[0])

title("Chocolate-Brainpower-Correlation: " + dec2(correlation(xval, yval)))
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

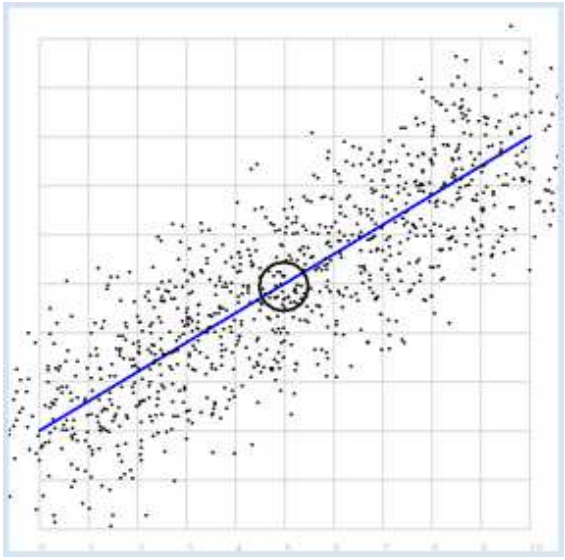
This results in a high correlation of approximately 0.71, which can be interpreted in different ways. It is right to say that there is a correlation between the two sets of data, but we can only speculate about the reasons why. In particular a causal relationship between the consumption of chocolate and intelligence can by no means be proven this way. As a general rule: When there is a high correlation between x and y, x can be the cause for the behavior of y. Likewise, y can be the cause for the behavior of x, or x and y can be associated with other perhaps unknown causes.

Discuss the meaning and purpose of this investigation with other people in your community and ask them for their opinion.

MEASUREMENT ERRORS AND NOISE

Even with an exact relationship between *x* and *y,* measurement errors or other influences can result in fluctuating measured values. In this simulation you assume a linear relationship between *x* and *y*, whereby the function values *y* are subjected to normally distributed fluctuations.

Determine the correlation coefficient and draw the point with the coordinates (*xm*, *ym*) where *xm* and *ym* are the expected values of *x* and *y*, respectively.



```
import random
from gpanel import *
import math

z = 1000
a = 0.6
b = 2
sigma = 1

def f(x):
    y = a * x + b
```

```
    return y

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
    return covariance(xval, yval) / (deviation(xval) * deviation(yval))

makeGPanel(-1, 11, -1, 11)
title("y = 0.6 * x + 2 normally distributed measurement errors")
addStatusBar(30)
drawGrid(0, 10, 0, 10, "gray")
setColor("blue")
lineWidth(3)
line(0, f(0), 10, f(10))

xval = [0] * z
yval = [0] * z

setColor("black")
for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, sigma)
    move(xval[i], yval[i])
    fillCircle(0.03)

xm = mean(xval)
ym = mean(yval)
move(xm, ym)
circle(0.5)

setStatusText("E(x) = " + dec2(xm) + \
          ",  E(y) = " + dec2(ym) + \
          ",  correlation = " + dec2(correlation(xval, yval)))
```

MEMO

This results in a high correlation as expected, which becomes even greater the smaller the selected dispersion *sigma* is. The correlation is exactly 1 for *sigma = 0*.
Also, the point with the expected values *P(0.5, 0.5)* lies on the straight line.

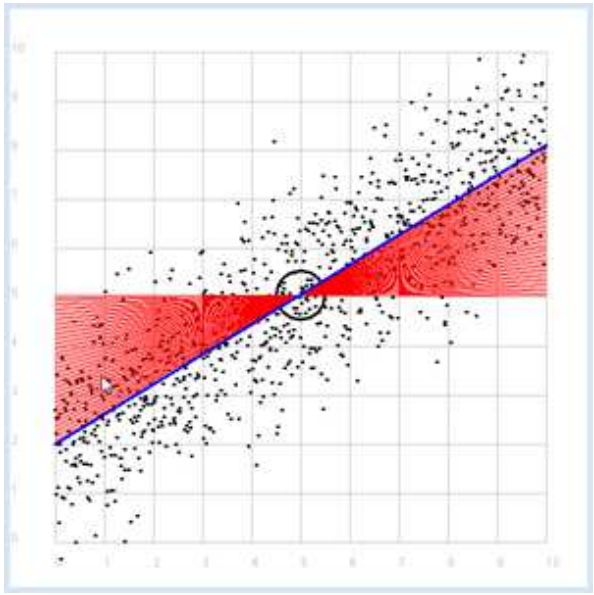
REGRESSION LINE, BEST FIT

Previously you started with a straight line, that you made "noisy" yourself. Here, you ask the opposite question: How do you detect the straight line again if you only have the two series of measurements? The desired line is called the **regression line**.

At least you already know that the regression line passes through the point P with the expected values. To find them, you can do the following:

You place any line through P and determine the squares of the deviations from the measured values for all *x* from the straight line (vertical distance). The deviations must be as small as possible for the regression line. For this, you determine an **error amount** by adding all individual deviations together.

You can find the best line in the simulation by gradually turning your placed line around the point P and always calculating the error sum, which will decline. You will have found the **best fit** just before the error sum begins to rise again.



```
import random
from gpanel import *
import math

z = 1000
a = 0.6
b = 2

def f(x):
    y = a * x + b
    return y

def dec2(x):
    return str(round(x, 2))

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
```

```
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

sigma = 1

makeGPanel(-1, 11, -1, 11)
title("Simulate data points. Press a key...")
addStatusBar(30)
drawGrid(0, 10, 0, 10, "gray")
setStatusText("Press any key")

xval = [0] * z
yval = [0] * z

for i in range(z):
    x = i / 100
    xval[i] = x
    yval[i] = f(x) + random.gauss(0, sigma)
    move(xval[i], yval[i])
    fillCircle(0.03)

getKeyWait()
xm = mean(xval)
ym = mean(yval)
move(xm, ym)
lineWidth(3)
circle(0.5)

def g(x):
    y = m * (x - xm) + ym
    return y

def errorSum():
    sum = 0
    for i in range(z):
        x = i / 100
        sum += (yval[i] - g(x)) * (yval[i] - g(x))
    return sum

m = 0
setColor("red")
lineWidth(1)
error_min = 0
while m < 5:
    line(0, g(0), 10, g(10))
    if m == 0:
        error_min = errorSum()
    else:
        if errorSum() < error_min:
            error_min = errorSum()
        else:
            break
    m += 0.01

title("Regression line found")
setColor("blue")
lineWidth(3)
line(0, g(0), 10, g(10))
setStatusText("Found slope: " +  dec2(m) + \
              ", Theory: " + dec2(covariance(xval, yval)
             /(deviation(xval) * deviation(xval))))
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

Instead of using a computer simulation to find the best fit, you can also directly calculate the slope of the regression line.

$$m = \frac{covariance(x, y)}{dispersion(x)^2}$$

Since the line passes through the point P with the expected values *E(x)* and *E(y)*, it thus has the linear equation:

$$y - E(y) = m * (x - E(x))$$

EXERCISES

1. The well-known Engel's law of economics states that the actual amount of money spent on food rises in households with a rising income, yet the proportion of income spent on food decreases. Show the accuracy of this using the data material.

   a. Visualize the relationship between income and total food expenditure (amount of money spent)

   b. Visualize the relationship between income and relative food expenditure

   c. Determine the correlation between income and absolute food expenditure

   d. Determine the regression line between income and absolute food expenditure

   Data:

Monthly income	4000	4100	4200	4300	4400	4500	4600	4700	4800	4900
Expenditure%	64	63.25	62.55	61.90	61.30	60.75	60.25	59.79	59.37	58.99

5000	5100	5200	5300	5400	5500	5600	5700	5800	5900	6000
58.65	58.35	58.08	57.84	57.63	57.45	57.30	57.17	57.06	56.97	56.90

2. The currently generally accepted evolutionary history of the universe assumes that there was a Big Bang a long time ago and since then, the universe expands. The main question is when the Big Bang dates back to, or what is called the **age of the universe**. The astronomer Hubble published his world famous studies in 1929, where he found that there is a linear relationship between the distance *d* of the galaxies and their escape velocity *v*. The Hubble law is:

   $$v = H * d$$

   (where H is the Hubble constant)

   You can comprehend the astrophysical thought process by starting with the following experimental data obtained from the Hubble Space Telescope:

Galaxy	Distance [Mpc)	Velocity [km/s]
NGC0300	2	133
NGC095	9.16	664
NGC1326A	16.14	1794
NGC1365	17.95	1594
NGC1425	21.88	1473
NGC2403	3.22	278

NGC2541	11.22	714
NGC2090	11.75	882
NGC3031	3.63	80
NGC3198	13.8	772
NGC3351	10	642
NGC3368	10.52	768
NGC3621	6.64	609
NGC4321	15.21	1433
NGC4414	17.7	619
NGC4496A	14.86	1424
NGC4548	16.22	1384
NGC4535	15.78	1444
NGC4536	14.93	1423
NGC4639	21.98	1403
NGC4725	12.36	1103
IC4182	4.49	318
NGC5253	3.15	232
NGC7331	14.72	999

1 Megaparsec (Mpc) = 3.09 * 10^{19}km

a) Plot the data in a scatter plot. Copy it to the data list into your program.

```
data = [
["NGC0300", 2.00, 133],
["NGC095", 9.16, 664],
["NGC1326A", 16.14, 1794],
["NGC1365", 17.95, 1594],
["NGC1425", 21.88, 1473],
["NGC2403", 3.22, 278],
["NGC2541", 11.22, 714],
["NGC2090", 11.75, 882],
["NGC3031", 3.63, 80],
["NGC3198", 13.80, 772],
["NGC3351", 10.0, 642],
["NGC3368", 10.52, 768],
["NGC3621", 6.64, 609],
["NGC4321", 15.21, 1433],
["NGC4414", 17.70, 619],
["NGC4496A", 14.86, 1424],
["NGC4548", 16.22, 1384],
["NGC4535", 15.78, 1444],
["NGC4536", 14.93, 1423],
["NGC4639", 21.98, 1403],
["NGC4725", 12.36, 1103],
["IC4182", 4.49, 318],
["NGC5253", 3.15, 232],
["NGC7331", 14.72, 999]]
```

from Freedman et al, The Astrophysical Journal, 553 (2001)

b) Show that the values correlate well and determine the slope H of the regression line.

c) If you assume that the velocity v of a certain galaxy remains constant, its distance is d = v * T, where T is the age of the universe. Following the Hubble law (v = H * d) we can deduce T = 1 / H . Determine T.

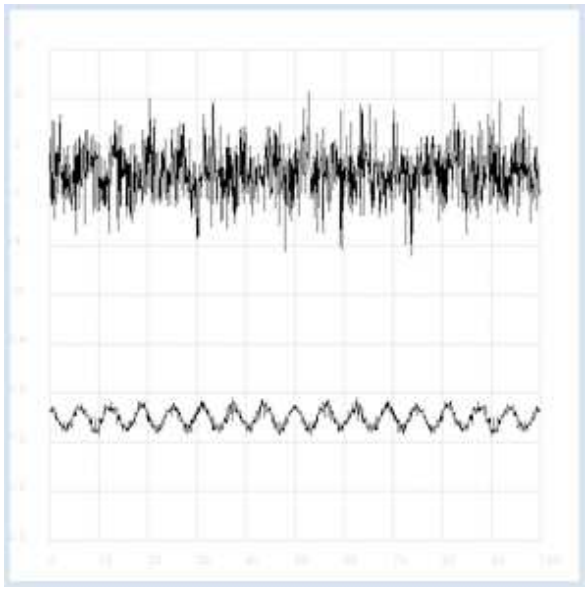
FINDING CACHED INFORMATION WITH AUTOCORRELATION

Intelligent beings on a distant planet want to contact other living beings. To do this, they send out radio signals that present a certain regularity (you can imagine a kind of Morse code). The signal becomes weaker and weaker on the long transmission path and gets masked by statistically fluctuating noise. We receive this signal on Earth with a radio telescope, and so for the time being we only hear the noise.

The statistics and a computer program can help us retrieve the original radio signal again. If you calculate the correlation coefficient of the signal with its own time-shifted signal, you decrease the statistical noise components. (A correlation with itself is called an **autocorrelation**).

You can simulate this property important for signal analysis with a *Python* program. The original useful signal is a sine wave and you superimpose it on the noise by adding a random number to each sample value (with a normal distribution). Show the noisy signal in the upper part of the graph and wait for a key press. The useful signal is no longer recognizable.

Subsequently, you build the autocorrelation of the signal and draw the course of the correlation coefficient in the lower part of the graph. The useful signal will be clearly recognizable again.



```
import random
from gpanel import *
import math

def mean(xval):
    n = len(xval)
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, yval):
    n = len(xval)
    xm = mean(xval)
    ym = mean(yval)
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (yval[i] - ym)
    return cxy / n

def deviation(xval):
    n = len(xval)
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

def correlation(xval, yval):
```

```
    return covariance(xval, yval)/(deviation(xval)*deviation(yval))
def shift(offset):
    signal1 = [0] * 1000
    for i in range(1000):
       signal1[i] = signal[(i + offset) % 1000]
    return signal1

makeGPanel(-10, 110, -2.4, 2.4)
title("Noisy signal. Press a key...")
drawGrid(0, 100, -2, 2.0, "lightgray")

t = 0
dt = 0.1
signal = [0] * 1000
while t < 100:
    y = 0.1 * math.sin(t) # Pure signal
#    noise = 0
    noise = random.gauss(0, 0.2)
    z = y + noise
    if t == 0:
        move(t, z + 1)
    else:
        draw(t, z + 1)
    signal[int(10 * t)] = z
    t += dt

getKeyWait()
title("Signal after autocorrelation")
for di in range(1, 1000):
    y = correlation(signal, shift(di))
    if di == 1:
        move(di / 10, y - 1)
    else:
        draw(di / 10, y - 1)
```

To make it a bit more exciting, listen to the noisy signal first and then the extracted useful signal. To do this, tap into your knowledge from the chapter on **Sound**.

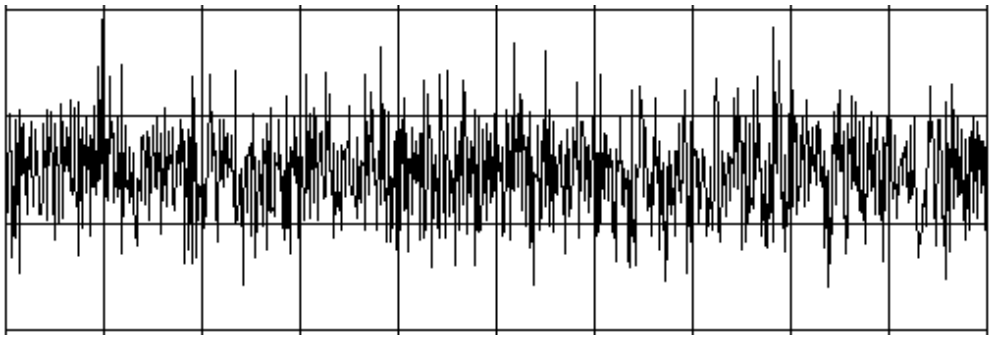
```
from soundsystem import *
import math
import random
from gpanel import *

n = 5000

def mean(xval):
    sum = 0
    for i in range(n):
        sum += xval[i]
    return sum / n

def covariance(xval, k):
    cxy = 0
    for i in range(n):
       cxy += (xval[i] - xm) * (xval[(i + k) % n] - xm)
    return cxy / n

def deviation(xval):
    xm = mean(xval)
    sx = 0
    for i in range(n):
        sx += (xval[i] - xm) * (xval[i] - xm)
    sx = math.sqrt(sx / n)
    return sx

makeGPanel(-100, 1100, -11000, 11000)
drawGrid(0, 1000, -10000, 10000)
title("Press <SPACE> to repeat. Any other key to continue.")
```

```
signal = []
for i in range(5000):
    value = int(200 *  (math.sin(6.28 / 20 * i) + random.gauss(0, 4)))
    signal.append(value)
    if i == 0:
        move(i, value + 5000)
    elif i <= 1000:
        draw(i, value + 5000)

ch = 32
while ch == 32:
    openMonoPlayer(signal, 5000)
    play()
    ch = getKeyCodeWait()

title("Autocorrelation running. Please wait...")
signal1 = []
xm = mean(signal)
sigma = deviation(signal)
q = 20000 / (sigma * sigma)
for di in range(1, 5000):
    value = int(q *  covariance(signal, di))
    signal1.append(value)
title("Autocorrelation Done. Press any key to repeat.")
for i in range(1, 1000):
    if i == 1:
        move(i, signal1[i] - 5000)
    else:
        draw(i, signal1[i] - 5000)

while True:
    openMonoPlayer(signal1, 5000)
    play()
    getKeyCodeWait()
```

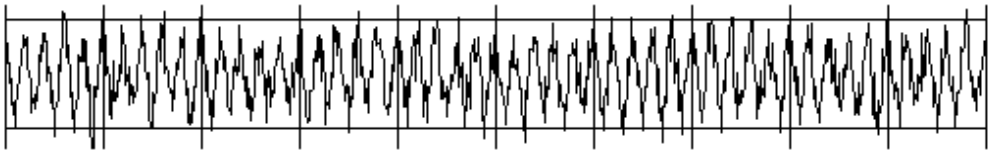
MEMO

Instead of executing the program, you can listen to the two signals as WAV files here

Noisy signal **(click here)**



Desired signal **(click here)**



8.7 COMPLEX NUMBERS & FRACTALS

INTRODUCTION

Complex numbers are very important in mathematics since they extend the set of real numbers allowing many propositions to be formulated easier and more general. They also play an important role in science and technology, especially in physics and electrical engineering [**more...** engineering, the alternating current theory is complex resistances (impedances) greatly simplified]. Fortunately, complex numbers are a built-in data type in *Python* and there are also arithmetic operators for addition, subtraction, multiplication, and division available for use. In addition, there are many known functions with complex arguments in the module *cmath*.

Complex numbers can be represented in the complex plane as arrows or as points. In order to allow that turtle windows, GPanels, and JGameGrid pixel grids can also be regarded as a complex planes, all functions with coordinate parameters (x,y) in the respective libraries are also directly available for complex numbers.

PROGRAMMING CONCEPTS: *Complex data type, conformal mapping, Mandelbrot fractal*

BASIC OPERATIONS WITH COMPLEX NUMBERS

In Python, for the imaginary unit you use the symbol *j* commonly seen in electrical engineering instead of *i*. You can define the complex number with the real part 2 and the imaginary part 3 in several ways:

```
z = 2 + 3j          or      z = 2 + 3 * 1j          or      z  = complex(2, 3)
```

Use the convenient *TigerJython* console for the following examples:

You get the real and imaginary parts by using *z.real* and *z.imag*, respectively. You should keep in mind that these are not function calls but rather variable values (which you can only read). Real and imaginary parts are always floats.

```
>>> z = 2 + 3j
>>> z
    (2+3j)
>>> z.real
    2.0
>>> z.imag
    3.0
```

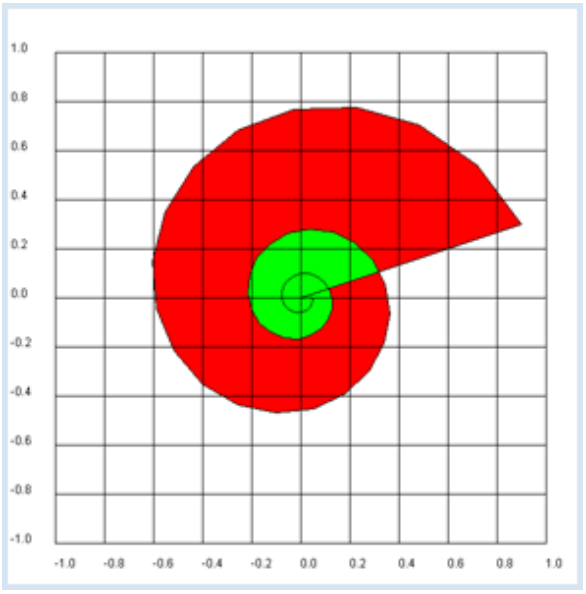
The square of the imaginary unit 1j is -1. In other words, the imaginary unit 1j is equal to the square root of -1. In order to get the square root of complex numbers, you have to import the module *cmath* instead of *math*

```
>>> z = 1j
>>> z * z
    (-1+0j)
>>> import cmath
>>> cmath.sqrt(-1)
    1j
```

The standard function *abs()* returns not only the value for integers and floats, but also for complex numbers. You can also use the usual operator symbols +, -, *, / and the power operator ** for complex numbers. They have the same order of precedence as for floats.

```
>>> z = 3 + 4j
>>> abs(z)
    5.0
>>> 2 * (z + 1) - z
    (5+4j)
>>> z**2 / z
    (3+4j)
```

In your program, you make powers of a complex number z = 0.9 + 0.3j, which has the absolute value of slightly less than 1. Since during the multiplication of two complex numbers the absolute values are multiplied and the phases added together , the powers seem to move on a spiral that you can easily draw in a *GPanel*. Remember that you have to do the filling before drawing the grid, because *fill()* always fills closed areas.




```
from gpanel import *
makeGPanel(-1.2, 1.2, -1.2, 1.2)
title("Complex plane")

z = 0.9 + 0.3j
for n in range(1, 60):
    y = z**n
    draw(y)
fill(0.2, 0, "white", "red")
fill(0.0, 0.2, "white", "green")
drawGrid(-1.0, 1.0, -1.0, 1.0)
```

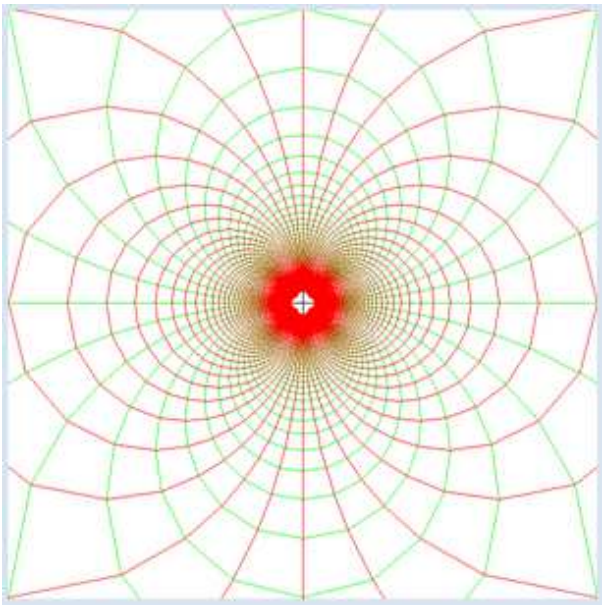
MEMO

draw(z) causes the same as *draw(z.real, z.imag)*, but it is simpler. You should use *GPanel* coordinates that are 10% larger on all sides than the used coordinate range. You must specify the coordinates as floats in *drawGrid()* so that the grid is labeled with floats.

CONFORMAL MAPPING

In a two-dimensional mapping every point *P(x, y)* is assigned a pixel *P'(x', y')*.

You can also regard the points as complex numbers and say that the mapping of each number z is assigned a function value z'. Therefore, you write z' = f(z).



In the following, you choose the function *z' = f(z) = 1/z* (inversion) and map a perpendicular coordinate grid.

You select the area from -5 to 5 in the complex plane and conceive 201 grid lines with a spacing of 1/20. Draw the horizontal lines green and the vertical lines red. The result is a pretty picture

```
from gpanel import *

# function f(z) = 1/z
def f(z):
```

```
    if z == 0:
        return 0
    return 1 / z

min = -5.0
max = 5.0
step = 1 / 20
reStep = complex(step, 0)
imStep = complex(0, step)

makeGPanel(min, max, min, max)
title("Conformal mapping for f(z) = 1 / z")
line(min, 0, max, 0)  # Real axis
line(0, min, 0, max) # Imaginary axis

# Transform horizontal line per line
setColor("green")
z = complex(min, min)
while z.imag < max:
    z = complex(min, z.imag) # left
    move(f(z))
    while z.real < max: # move along horz. line
        draw(f(z))
        z = z + reStep
    z = z + imStep

# Transform vertical line per line
setColor("red")
z = complex(min, min)
while z.real < max:
    z = complex(z.real, min) # bottom
    move(f(z))
    while z.imag < max:  # move along vert. line
        draw(f(z))
        z = z + imStep
    z = z + reStep
```

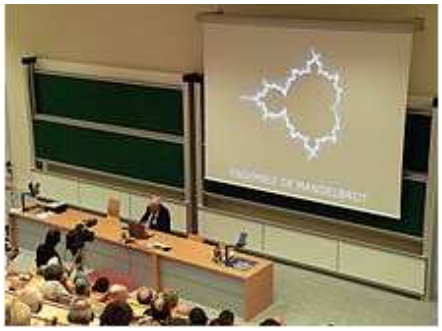
Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

What you can gather from the picture is that the images of the grid lines still intersect perpendicularly. A mapping where the angles between intersecting lines is maintained is called **conformal** and so we speak of a **conformal mapping** [**more...**] .

MANDELBROT FRACTALS

Many people are familiar with fractal images. It is likely that you have already encountered the **Mandelbrot set**, which you will program here yourself. The algorithm for many fractals is based on complex numbers, and because of this, it is very rewarding to work with them. The mathematician Benoit Mandelbrot is the father of **fractal geometry** (1924-2010).



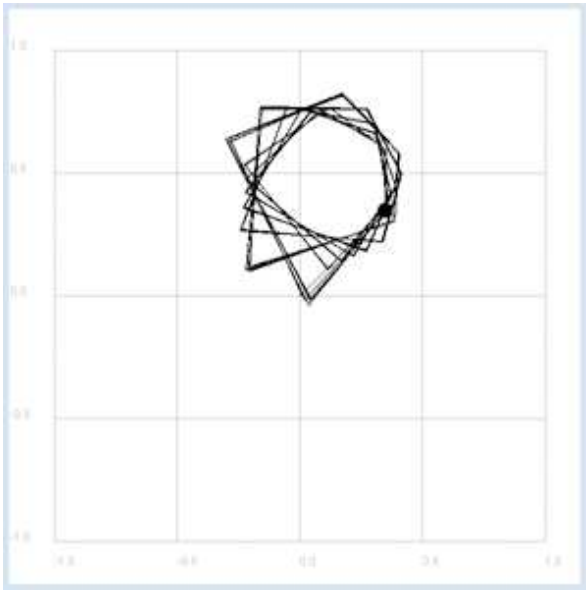
Mandelbrot at the introductory lecture of the Légion d'honneur (2006) (© Wiki)

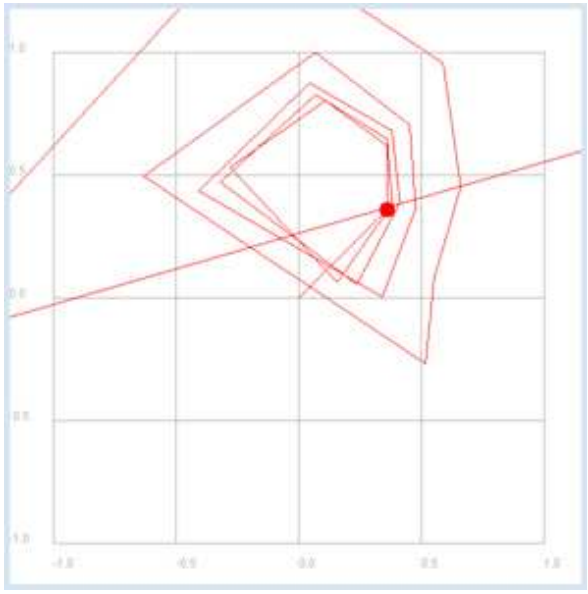
To generate a Mandelbrot fractal you look at a (recursively defined) sequence of complex numbers for the given complex number *c* according to the formation rule

$z' = z^2 + c$ with the initial value $z_0 = 0$

If the amount of the sequence terms remains in a confined area, meaning that they do not grow beyond all limits, *c* belongs to the **Mandelbrot set**.

To familiarize yourself with the algorithm, you draw the following terms for two complex numbers $c_1 = 0.35 + 0.35j$ and $c_2 = 0.36 + 0.36j$. You will immediately see that c1 belongs to the Mandelbrot set, but c2 does not.

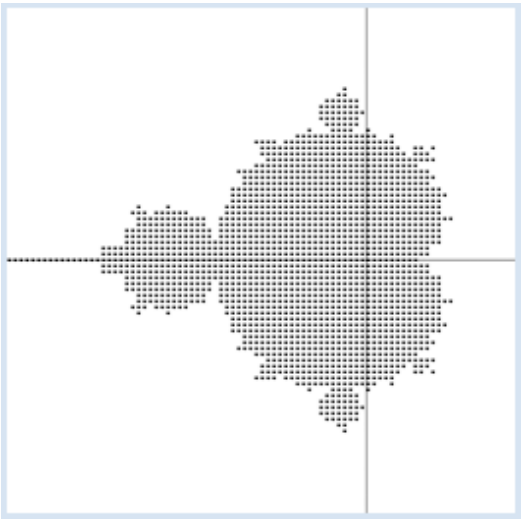




```
from gpanel import *

def f(z):
    return z * z + c

makeGPanel(-1.2, 1.2, -1.2, 1.2)
title("Mandelbrot iteration")

drawGrid(-1, 1.0, -1, 1.0, 4, 4, "gray")

isMandelbrot = askYesNo("c in Mandelbrot set?")
if isMandelbrot:
    c = 0.35 + 0.35j
    setColor("black")
else:
    c = 0.36+ 0.36j
    setColor("red")

title("Mandelbrot iteration with c = " + str(c))
move(c)
fillCircle(0.03)

z = 0j
while True:
    if z == 0:
        move(z)
    else:
        draw(z)
    z = f(z)
    delay(100)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

To find out which numbers in a certain area of the complex plane belong to the Mandelbrot set, you execute the iteration for complex numbers *c* in a given range of the grid. Here you very simplistically assume that *c* does not belong to the Mandelbrot set if the absolute value of a sequence term in the first 50 iterations is greater than R = 2. If the absolute value of *z* remains less than 2 until the end of the 50 iterations, you assume that *c* belongs to the Mandelbrot set and you draw a black point there.

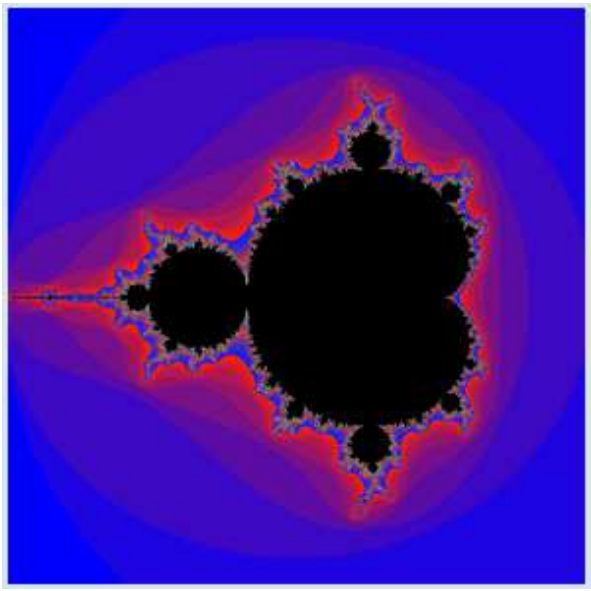


```
from gpanel import *

def isInSet(c):
    z = 0
    for n in range(maxIterations):
        z = z*z + c
        if abs(z) > R: # diverging
            return False
    return True

maxIterations = 50
R = 2
xmin = -2
xmax = 1
xstep = 0.03
ymin = -1.5
ymax = 1.5
ystep = 0.03

makeGPanel(xmin, xmax, ymin, ymax)
line(xmin, 0, xmax, 0)  # real axis
line(0, ymin, 0, ymax) # imaginary axis
title("Mandelbrot set")
y = ymin
while y <= ymax:
    x = xmin
    while x <= xmax:
        c = x + y*1j
        inSet = isInSet(c)
        if inSet:
            move(c)
            fillCircle(0.01)
        x += xstep
    y += ystep
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

The Mandelbrot set is already visible in the graphic. You can draw figures that are even more beautiful if you draw a colored point for numbers c that do **not** belong to the Mandelbrot set, where the color says how quickly the sequence diverges. In this case, our measure of divergence is the number *itCount,* which represents the number of iterations after which the amount of *z* is greater than *R = 2* for the first time.

To map *itCount* to a color, you can decide on any combination in *getIterationColor()* that in your opinion creates a particularly beautiful fractal.



```
from gpanel import *

def getIterationColor(it):
    color = makeColor((30 * it) % 256,
                      (4 * it) % 256,
                      (255 - (30 * it)) % 256)
    return color

def mandelbrot(c):
    z = 0
    for it in range(maxIterations):
        z = z*z + c
        if abs(z) > R: # diverging
            return it
    return maxIterations

maxIterations = 50
R = 2
xmin = -2
xmax = 1
xstep = 0.003
ymin = -1.5
ymax = 1.5
ystep = 0.003

makeGPanel(xmin, xmax, ymin, ymax)
title("Mandelbrot set")
enableRepaint(False)
y = ymin
while y <= ymax:
    x = xmin
    while x <= xmax:
        c = x + y*1j
        itCount = mandelbrot(c)
        if itCount == maxIterations: # inside Mandelbrot set
            setColor("black")
        else: # outside Mandelbrot set
           setColor(getIterationColor(itCount))
        point(c)
        x += xstep
    y += ystep
    repaint()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

To speed up the drawing, you set *enableRepaint(False)* and only re-render at the end of each line using *repaint()*.

The Mandelbrot fractal possesses the remarkable feature that when a section is magnified, a similar structure appears [**more...**] .

EXERCISES

1. You can create a beautiful fractal if in a grid of the complex plane in the area between -20 and 20 you draw only the grid points *z*, for which the rounded absolute value is even, so *int(abs(z) * abs(z)) % 2 == 0* applies. Choose an increment size of 0.1.

2. Analyze the mappings of the complex plane:

   a) $z' = f(z) = z^2$

   b) $z' = f(z) = a * z$ mit komplexem $a = 2 + 1j$

   c) $z' = f(z) = e^z$

   d) $z' = f(z) = \frac{1 - z}{1 + z}$ (Möbius-Transformation)

   Map a rectangular coordinate grid in the range between -5 and 5 with an increment value of 1/10. Describe the image with words and speculate on whether it is conformal.

3. Draw some Mandelbrot fractals with different color mappings, for example:

Number of iterations	**Color**
< 3	dark gray
< 5	green
< 8	red
< 10	blue
< 100	yellow
sonst	black

ADDITIONAL MATERIAL

ALTERNATING CURRENT AND IMPEDANCE

Electrical circuits for sinusoidal alternating voltages and alternating currents, which are built from passive devices (resistors, capacitors, inductors), can be treated like direct current circuits if you are using complex variables for voltages, currents, and resistors. A general complex resistor is also called an **impedance** and is often denoted with Z (and for purely imaginary resistors, X). The impedance of an ohmic resistor is R, of an inductor is $X_L = j\omega L$ (L: inductance) and of a capacitor is $X_C = 1 / j\omega C$ (C: capacitance), where $\omega = 2\pi f$ (f: frequency). A complex alternating voltage u = u(t) runs uniformly on a circle in the complex plane. If it is applied to an impedance Z, the current i(t) flows according to Ohm's law u = Z * i. Since in complex multiplication the phases are added and the absolute values multiplied, *u* runs before i, phase-shifted by the phase

of Z

phase(u) = phase(Z) + phase(i)

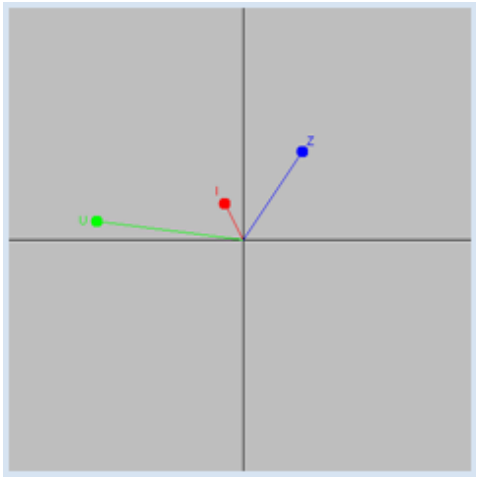
So, *i* also runs on a circle. The following absolute values (amplitudes) we have:

*| u | = | Z | * | i |*

In your program, you display these relationships in the complex plane with the values

| u | = 5V and *Z = 2 +3j*

and a frequency of f = 10 Hz. Since the graphic is completely erased, rebuilt, and rendered with *repaint()* in each step of the animation, use a *GPanel* with *enableRepaint(False)*.

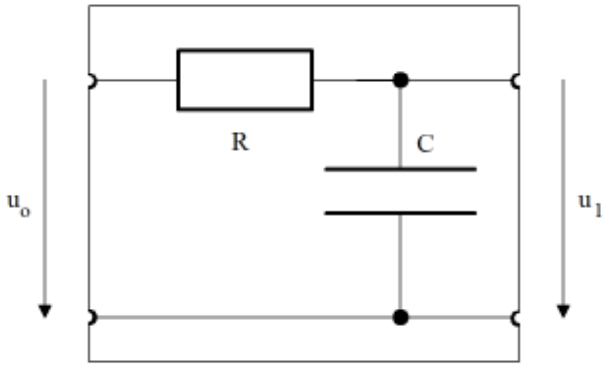



```
from gpanel import *
import math

def drawAxis():
   line(min, 0, max, 0)  # real axis
   line(0, min, 0, max) # imaginary axis


def cdraw(z, color, label):
    oldColor = setColor(color)
    line(0j, z)
    fillCircle(0.2)
    z1 = z + 0.5 * z / abs(z) - (0.1 + 0.2j)
    text(z1, label)
    setColor(oldColor)

min = -10
max = 10
dt = 0.001

makeGPanel(min, max, min, max)
enableRepaint(False)
bgColor("gray")
title("Complex voltages and currents")

f = 10 # Frequency
omega = 2 * math.pi * f

t = 0
uA = 5
Z = 2 + 3j

while True:
    u = uA * (math.cos(omega * t) + 1j * math.sin(omega * t))
    i = u / Z
    clear()
    drawAxis()
    cdraw(u, "green", "U")
    cdraw(i, "red", "I")
    cdraw(Z, "blue", "Z")
    repaint()
    t += dt
    delay(100)
```

MEMO

Electrical circuits with passive devices can be treated as direct current circuits if you regard voltage, current, and resistance as complex numbers.

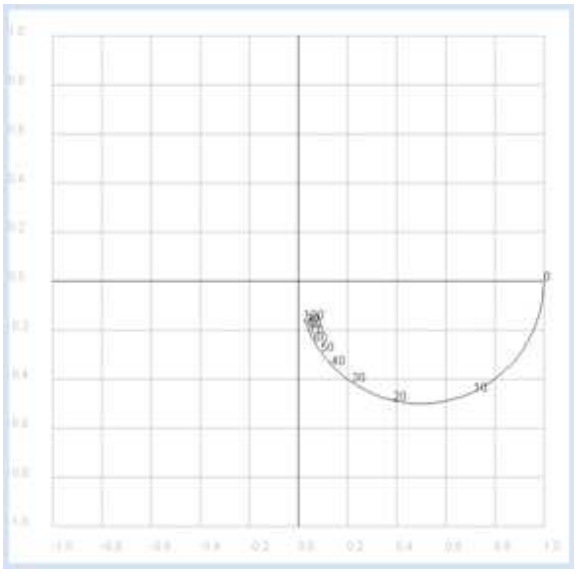
You can apply this knowledge to a simple circuit consisting of only a resistor and a capacitor. You are interested in finding the output voltage u1 dependent on the frequency *f*, regarding the input voltage u_o, R and C as given.



The calculation is simple: The series circuit of R and C results in the impedance Z = R + X_C and thus the current i = u_o / Z, so again using Ohm's law the output voltage

$$u_1 = X_c * i = \frac{X_c}{R + X_c} * u_0 = \quad \text{or} \quad u_1 = v * u_0 \quad \text{with} \quad v = \frac{X_c}{R + X_c}$$

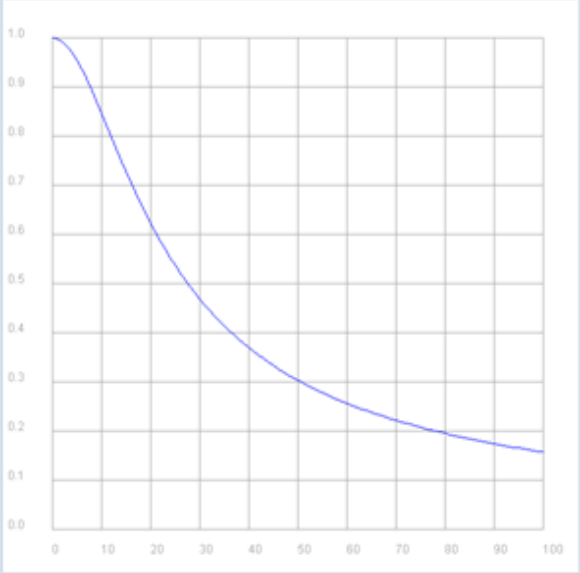
v is called the complex **amplification factor**. You can visualize it in the complex plane for different values of *f* and you see that the amount of the value 1 decreases with increasing frequency at the frequency 0. Low frequencies are thus transferred well, whereas high frequencies are transferred poorly. This circuit would therefore be a **low pass filter**.

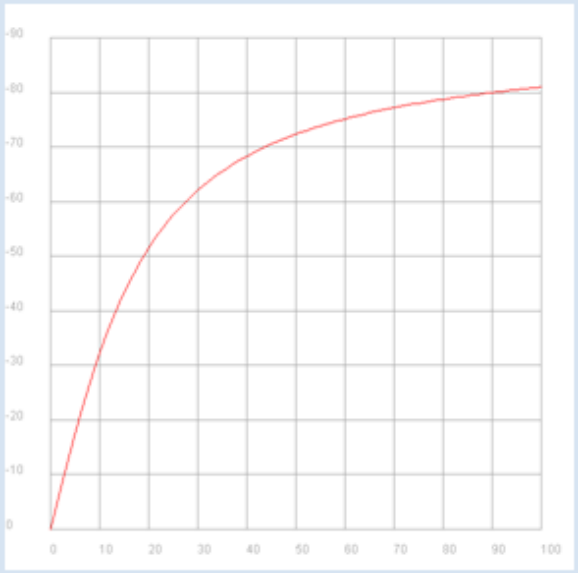


```
from gpanel import *
from math import pi

def drawAxis():
   line(-1, 0, 1, 0)  # Real axis
   line(0, -1, 0, 1) # Imaginary axis

makeGPanel(-1.2, 1.2, -1.2, 1.2)
drawGrid(-1.0, 1.0, -1.0, 1.0, "gray")
setColor("black")
drawAxis()
title("Complex gain factor – low pass")

R = 10
C = 0.001
def v(f):
    if f == 0:
        return 1 + 0j
```

```
    omega = 2 * pi * f
    XC = 1 / (1j * omega * C)
    return XC / (R + XC)

f = 0 # Frequency
while f <= 100:
    if f == 0:
        move(v(f))
    else:
        draw(v(f))
    if f % 10 == 0:
        text(str(f))
    f += 1
    delay(10)
```

The gain factor in the **Bode plot** is divided by magnitude and phase, and plotted in function of the frequency (logarithmic scales are commonly used).






```
from gpanel import *
import math
import cmath

R = 10
C = 0.001

def v(f):
    if f == 0:
        return 1 + 0j
    omega = 2 * math.pi * f
    XC = 1 / (1j * omega * C)
    return XC / (R + XC)

p1 = GPanel(-10, 110, -0.1, 1.1)
drawPanelGrid(p1, 0, 100, 0, 1.0, "gray")
p1.title("Bode Plot - Low Pass, Gain")
p1.setColor("blue")
f = 0
while f <= 100:
    if f == 0:
        p1.move(f, abs(v(f)))
    else:
        p1.draw(f, abs(v(f)))
    f += 1

p2 = GPanel(-10, 110, 9, -99)
drawPanelGrid(p2, 0, 100, 0, -90, 10, 9, "gray")
```

```
p2.title("Bode Plot - Low Pass, Phase")
p2.setColor("red")
f = 0
while f <= 100:
    if f == 0:
        p2.move(f, math.degrees(cmath.phase(v(f))))
    else:
        p2.draw(f, math.degrees(cmath.phase(v(f))))
    f += 1
```

MEMO

The Bode-Plot once again particularly clarifies that the present circuit transmits low frequencies well and high frequencies poorly. Additionally, a phase shift exists between the input and the output signal in the range of 0 to -90 degrees. One could also say that the output voltage "lags" behind the input voltage or that the input voltage "leads" the output voltage.

EXERCISES

1. The frequency

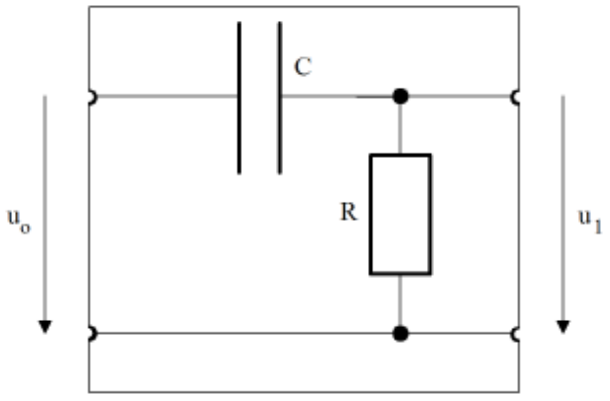
$$f_c = \frac{1}{2\,\pi\,R\,C}$$

is called the cutoff frequency. Show that the amount of the gain factor for the RC low pass filter when R = 10 Ohm and C = 0.001 F at this frequency is:
$1 / \sqrt{2}$

2. The amount of the gain factor is often specified in decibels (dB). It is defined dB = 20 log |v| (decimal logarithm). Draw the Bode diagram for the RC low pass filter with R = 10 Ohm and C = 0.001 F with a dB scale up to -100 dB and a logarithmic frequency scale in the range 1Hz..100 kHz.

   Confirm that the reduction for higher frequencies amounts to 20 dB/(frequency decade) using the graphic illustration.

3. The following circuit is a **high pass filter** (R = 10 Ohm, C = 0.001 F).



   As you did in exercise 2, draw the Bode plot for the gain factor and discuss the frequency behavior.

8.8 SPECTRAL ANALYSIS

INTRODUCTION

When a beam of light falls on your eyes or you hear a tone, a signal accrues that can be regarded as a function of time *y(t)*. With a single spectral color or a pure tone, it consists of a sinusoidal oscillation with the amplitude *A* and frequency *f*, expressed mathematically [**more...**]:

$$y(t) = A\sin(\omega * t) \text{ where } \omega = 2 * \pi * f$$

A more complex signal, for example by a constant sustained note of a musical instrument, is still periodic but no longer sinusoidal. The famous mathematician Joseph Fourier (1768-1830) proved that one can also interpret each periodic function as a sum of sine oscillations, as a so-called **Fourier series**. He thus laid an invaluable foundation for the progress of modern mathematics, physics, and engineering. Breaking a signal down into its sinusoidal frequency components is called **spectral analysis**..

PROGRAMMING CONCEPTS: *Sine oscillation, Fourier series, Fast Fourier Transform (FFT), spectrum, sonogram*

SPECTRUM OF A SOUND, OVERTONES

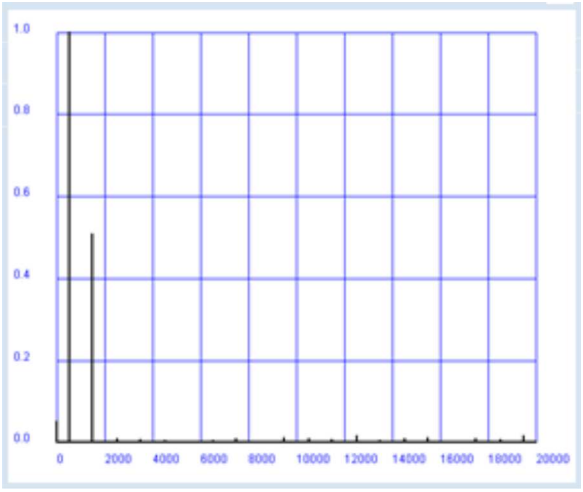
The sinusoidal frequency components that are present are important for the typical tone color of a voice or a musical instrument. A strictly periodic sound consists of the fundamental and the overtones, whose frequencies are integer multiples of the frequency of the fundamental. If you plot the amplitude of the frequency components in a graph, you get the **spectrum of the sound**. You can determine the spectrum with a device called a **spectral analyzer**. *TigerJython* can determine the spectrum using a famous algorithm called the **Fast Fourier Transform** (FFT).

In order to perform the FFT, you give the function *fft(samples, n)* a list *samples* that contains the temporally equidistant sample values and the number *n* of samples that should be used for the FFT from the beginning of the list.

As return values, you get back a list with the amplitudes of the *n/2* (normalized) frequency components. These are separated by the distance *r = fs / n,* where *fs* is the sampling frequency. *r* is (called) the **resolution** of the spectrum.

These *n/2* return values at the distance *r* "populate" the frequency range from 0 to *n/2*r = fs/2*, or in a nutshell: **The FFT provides the spectrum from 0 to *fs/2* at a sampling frequency of *fs***. A CD with a typical sampling frequency where *fs* = 44100 Hz corresponds to a spectrum up to 22050 Hz, which covers the entire audible range of humans.

In order to test the spectrum analyzer, you first use a sound from the distribution of *TigerJython* named *"wav/doublesine.wav"* that superimposes two sine tones. The sound was recorded at a sampling frequency of *fs* = 40,000 Hz. If you take *n* = 10,000 sampling values, the function *fft(samples, n)* returns 5'000 frequency components with the resolution *r* = 40,000 / 10,000 = 4 Hz in the range from 0..20,000 Hz, which you can then display graphically as a spectrum with vertical lines in a *GPanel*.



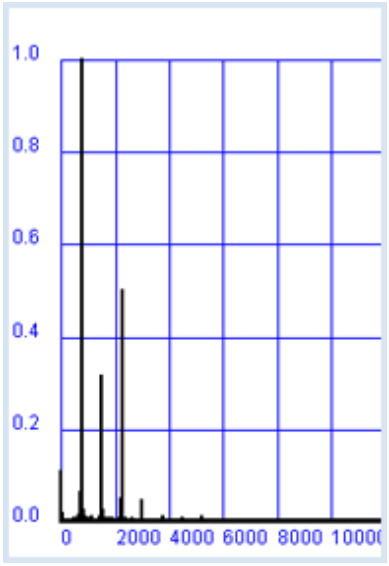

```
from soundsystem import *
from gpanel import *

def showSpectrum(text):
    makeGPanel(-2000, 22000, -0.2, 1.2)
    drawGrid(0, 20000, 0, 1.0, 10, 5, "blue")
    title(text)
    lineWidth(2)
    r = fs / n # Resolution
    f = 0
    for i in range(n // 2):
        line(f, 0, f, a[i])
        f += r

fs = 40000 # Sampling frequency
n = 10000 # Number of samples
samples = getWavMono("wav/doublesine.wav")
openMonoPlayer(samples, fs)
play()
a = fft(samples, n)
showSpectrum("Audio Spectrum")
```

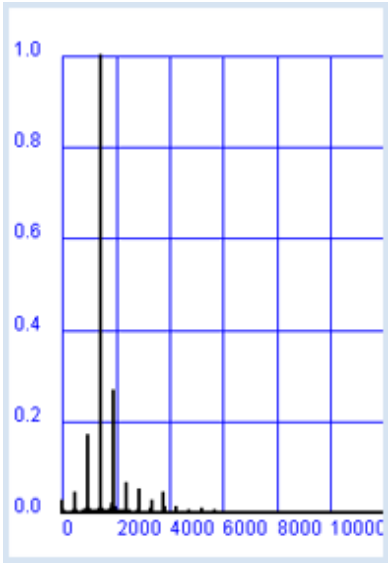
Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

As you imagine (and hear) you find the frequencies 500 Hz and 1.5 kHz with an amplitude ratio of 1 : 1/2. There are some additional disturbance components. The frequency 0 corresponds to a constant signal component (offset).

You now have a feudal spectrum analyzer in front of you, with which you can examine the fundamentals and overtones of musical instruments, human sounds, or animal sounds with. You will already find the sound of a flue ("wav/flute.wav") and an oboe ("wav/oboe.wav") in the distribution, whose sound characteristics are very different.

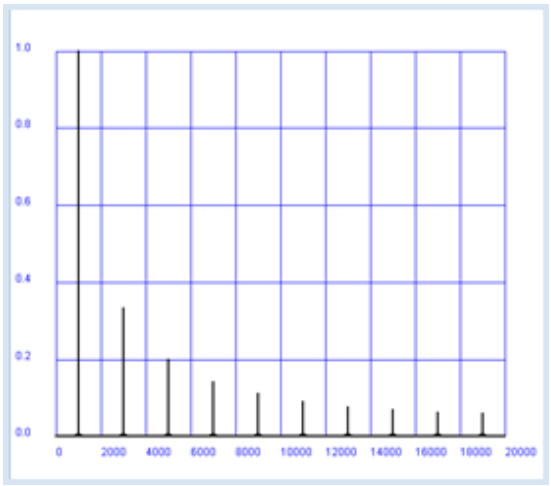




SPECTRA FOR SELF-DEFINED FUNCTIONS

According to the theorem of Fourier, every periodic function with the frequency *f* can be represented as superpositions of sine functions with the frequencies f, 2*f, 3*f, etc. (Fourier series).

You can experimentally determine the amplitudes of these frequency components with your Fourier analyzer. Here you consider a square wave with the frequency *f* = 1 kHz. The built-in function *square(A, f, t)* provides you with the value *A* during the first half of the period and *-A* in the second.



You choose a sampling frequency of *fs* = 40 kHz and determine the sound samples for a duration of 3s (120'000 values). Then you play the sound clip. For the spectrum, however, you only use 10,000 values and display it.

```
from soundsystem import *
from gpanel import *

def showSpectrum(text):
    makeGPanel(-2000, 22000, -0.2, 1.2)
    drawGrid(0, 20000, 0, 1.0, 10, 5, "blue")
    title(text)
    lineWidth(2)
    r = fs / n # Resolution
    f = 0
    for i in range(n // 2):
        line(f, 0, f, a[i])
        f += r

n = 10000
fs = 40000 # Sampling frequency
f = 1000 # Signal frequency

samples = [0] * 120000  # sampled data for 3 s
t = 0
dt = 1 / fs # sampling period
for i in range(120000):
   samples[i] = square(1000, f, t)
```

```
    t += dt

openMonoPlayer(samples, 40000)
play()
a = fft(samples, n)
showSpectrum("Spectrum Square Wave")
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The experiment shows that the spectrum of a rectangular function consists of the odd multiples of the fundamental frequency and where the amplitudes of the spectral components behave as 1, 1/3, 1/5, 1/7, etc. However, you will never be able to find out experimentally that the spectral parts theoretically stretch until ad infinitum.

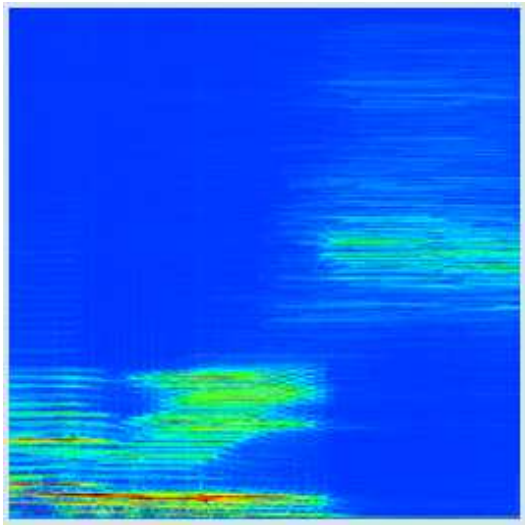
SONOGRAM

FFT is a perfect tool to record the spectral behavior of a sound varying in time, such as a spoken word. Of course in this case, the signal is no longer periodical, but you can assume that it is somewhat periodic piecewise. That is why FFT is often used for short signal blocks, for example for a block length of 100 ms, and repeated every 2.5 ms. Hence, we get a new spectrum for every 2.5 ms that can be represented as a colored vertical line in a sonogram.

In your program, you start at the beginning of the sampling values and analyze a block length of 2000 values. You begin the next block 50 samples later, etc. In Python, you can do this with a slice operation

samples[k * 50:] where k = 0, 1, 2,...

This results in a **sonogram,** for example for the spoken word "harris", located in the distribution of *TigerJython* as "*wav/harris.wav*".



```
from soundsystem import *
from gpanel import *

def toColor(z):
    w = int(450 +  300 * z)
    c = X11Color.wavelengthToColor(w)
    return c

def drawSonogram():
    makeGPanel(0, 190, 0, 1000)
    title("Sonogramm of 'Harris'")
    lineWidth(4)
    # Analyse blocks every 50 samples
    for k in range(191):
        a = fft(samples[k * 50:], n)
        for i in range(n // 2):
            setColor(toColor(a[i]))
            point(k, i)
```

```
fs = 20000 # Sampling freq->spectrum 0..10 kHz
n = 2000 #  Size of block for analyser

samples = getWavMono("wav/harris.wav")
openMonoPlayer(samples, fs)
play()
drawSonogram()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

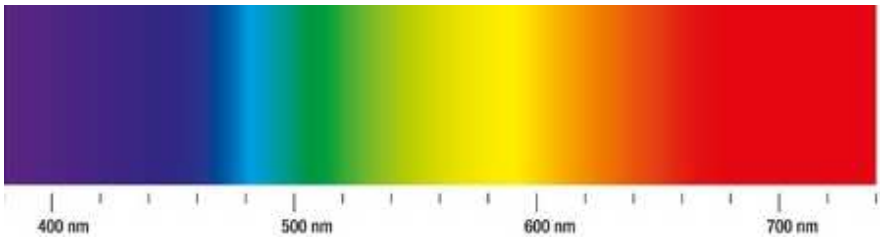
The sonogram shows frequencies in the range from 0..10 kHz vertically, and the course of time from 0 to 190 * 50 / 20000 = 0.475 s horizontally.

To convert numbers to colors, you should use the function *X11Color.wavelengthToColor*() which converts the wavelengths of the color spectrum to colors in the range 380...780 nm.

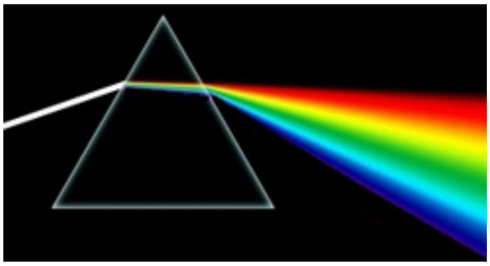
The high spectral components for the sibilant "s" are clearly visible, whereas the fundamentals are entirely missing.

LIGHT SPECTRA

Light can also be decomposed spectrally in order to determine its contained wavelength components. The wavelengths of the visible spectrum ranges from about 380 nm to 780 nm.



It is quite likely that you already know the spectrum analyzer for light, called a prism, which refracts (breaks up) light of various wavelengths at different angles according to the law of refraction.



In your program, you simulate the transition of a white beam of light in glass and show in a magnification the paths of the different colors.

```
from gpanel import *

# K5 glass
B = 1.5220
C = 4590  # nanometer^2
# Cauchy equation for refracting index
def n(wavelength):
    return B + C / (wavelength * wavelength)

makeGPanel(-1, 1, -1, 1)
title("Refracting at the K5 glass")
bgColor("black")
```

```
setColor("white")
line(-1, 0, 1, 0)


lineWidth(4)
line(-1, 1, 0, 0)
lineWidth(1)

sineAlpha = 0.707

for i in range(51):
     wavelength = 380 + 8 * i
     setColor(X11Color.wavelengthToColor(wavelength))
     sineBeta = sineAlpha / n(wavelength)
     x = (sineBeta - 0.45) * 100 - 0.5  # magnification
     line(0, 0, x, -1)
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

If you want to create a beautiful graphic, you should refract the colors more than they would be in real life.

EXERCISES

1. Examine other instruments or voices regarding their overtones and try to understand your findings based on the typical character of the sound.

2. Besides the global function *square(A, f, t),* the functions *sine(a, f, t), triangle(A, f, t), sawtooth(A, f, t)* are available. Try to guess how the spectrum of a triangular and a sawtooth wave is. You can also analyze the superpositions of sine waves using *sine()*.

3. In sonograms, analyze the difference between various female and male speakers who say the same word.

8.9 GROUP DYNAMICS

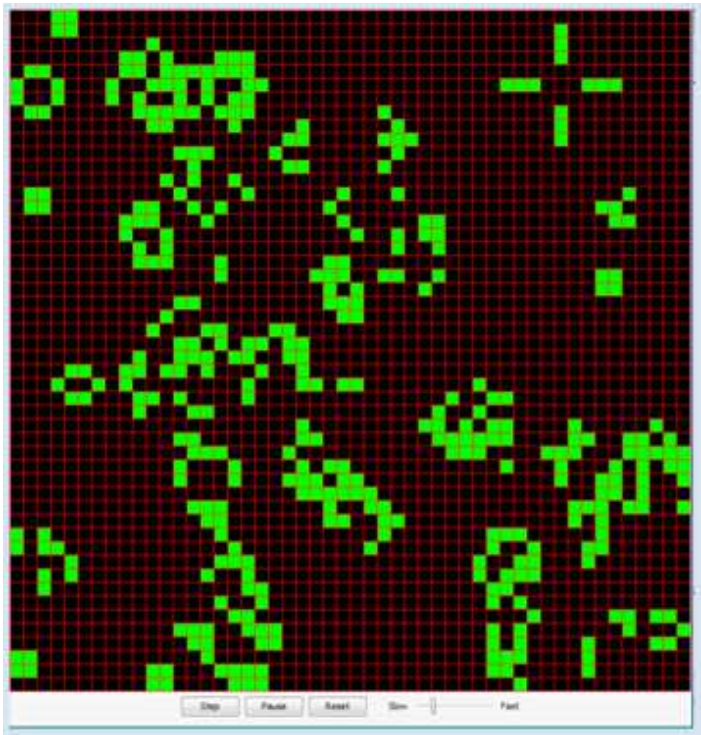
INTRODUCTION

Systems with many partners that interact with each other are widely used. Computer programs can often simulate such systems with surprisingly little effort, since the computer can store anywhere from thousands to millions of single individual states and temporally track them. However, if we deal with atomic many-body systems with a numbers of particles in the order of 10^{23}, computers reach their limits. To simulate such systems, we need to use simplifying procedures, such as dividing the system into several larger cells. Examples for this include the simulation of Earth's atmosphere for weather forecast and the prediction of long-term climate change.

PROGRAMMING CONCEPTS: *Computer simulation, population dynamics, swarming behavior*

CONWAY'S GAME OF LIFE

Conway's Game of Life examines a two-dimensional grid-like arrangement of individuals (green squares), where each individual interacts with its 8 nearest neighbors. It was proposed by the British mathematician John Conway in 1970, and it made him famous outside of the mathematics world as well. Almost all scholars have at least an idea of what the Game of Life is. Here you will program it yourself with *Python*.



The population is arranged in cells and evolves in discrete time steps (generations). Each cell can be either *living* or *dead*.

When transitioning to the next generation, the current state is saved and the following state of each cell is determined based on its 8 nearest neighbors by the following four transition rules:

1. If the cell is living, it dies if it has fewer than two living neighbors (isolation)
2. If the cell is living, it continues to live if it has two or three living neighbors (group cohesion)
3. If the cell lives, it dies if it has more than three living neighbors (overpopulation)
4. If a cell is dead, it will come back to life when it has exactly three living neighbors (reproduction). Otherwise, it stays dead.

The cell structure of *GameGrid* is ideal for implementing the game. You use a two-dimensional list *a[x][y]* for the population, where the value 0 is a dead cell and 1 is a living cell. The

simulation cycle is regarded as a generation cycle, and the current population from the list *a* is copied into the new population *b* in the callback *onAct()* and will finally be regarded as the current list. You choose 1,000 random living cells in the callback *onReset()*, which is called by clicking on the reset button.

In order to activate the callbacks, you have to register them with *registerAct()* and *registerNavigation()*..

```
from gamegrid import *

def onReset():
    for x in range(s):
        for y in range(s):
            a[x][y] = 0  # All cells dead
    for n in range(z):
        loc = getRandomEmptyLocation()
        a[loc.x][loc.y] = 1
    showPopulation()

def showPopulation():
    for x in range(s):
        for y in range(s):
            loc = Location(x, y)
            if a[x][y] == 1:
                getBg().fillCell(loc, Color.green, False)
            else:
                getBg().fillCell(loc, Color.black, False)
    refresh()

def getNumberOfNeighbours(x, y):
    nb = 0
    for i in range(max(0, x - 1), min(s, x + 2)):
        for k in range(max(0, y - 1), min(s, y + 2)):
            if not (i == x and k == y):
                if a[i][k] == 1:
                    nb = nb + 1
    return nb

def onAct():
    global a
    # Don't use the current, but a new population
    b  = [[0 for x in range(s)] for y in range(s)]
    for x in range(s):
        for y in range(s):
            nb = getNumberOfNeighbours(x, y)
            if a[x][y] == 1:  # living cell
                if nb < 2:
                    b[x][y] = 0
                elif nb > 3:
                    b[x][y] = 0
                else:
                    b[x][y] = 1
            else:             # dead cell
                if nb == 3:
                    b[x][y] = 1
                else:
                    b[x][y] = 0
    a = b # Use new population as current
    showPopulation()

# ================== global section ==================
s = 50   # Number of cells in each direction
z = 1000 # Size of population at start
a  = [[0 for x in range(s)] for y in range(s)]
makeGameGrid(s, s, 800 // s, Color.red)
registerAct(onAct)
registerNavigation(resetted = onReset)
setTitle("Conway's Game Of Life")
```

```
onReset()
show()
```

MEMO

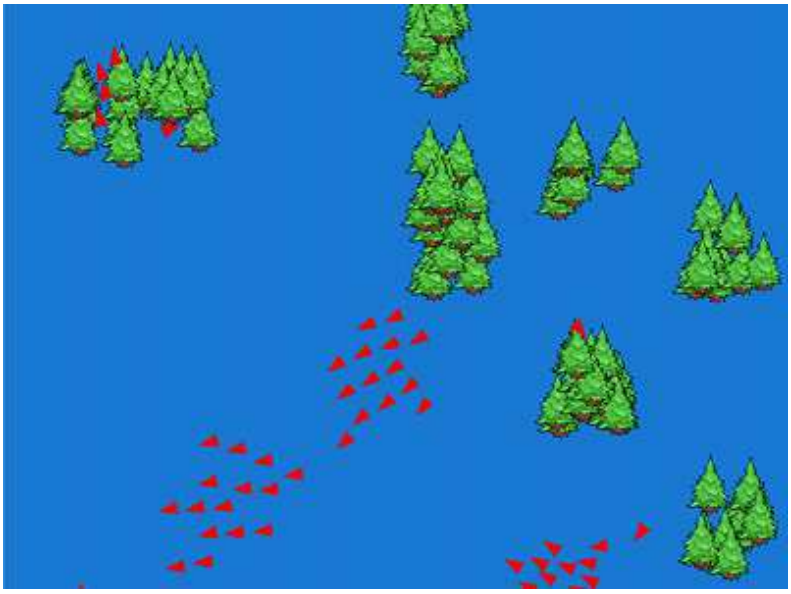
The Game of Life is an example of a **cellular automaton**, consisting of grid cells that interact with each other. They are perfectly suited to study the behavior of complex natural systems. Some examples are:

- biological growth, emergence of life
- social, geological, ecological behavior
- traffic volume and control
- formation and evolution of the cosmos, of galaxies and stars

In 2002, Stefan Wolfram, the scientist and chief developer of Mathematica pointed out in his well known book "A New Kind of Science" that such systems can be investigated with simple programs. With computer simulations, we are at the beginning of a new era of gaining scientific knowledge.

During the initialization of the two-dimensional list, a special Python syntax called **list comprehension** is used (see **addional material**).

SWARMING BEHAVIOR



As you know from your daily life, large groups of living beings often have the tendency to team up together in groups. This can be observed particularly well in birds, fish, and insects. A group of demonstrating people also shows this "swarming behavior". On the one hand, outside (global) influences play a role in the formation of a swarm, but the interaction between partners in their close surroundings (local influences) also play a role.

In 1986 Craig Reynolds showed that the following three rules lead to a swarm formation between individuals (he called them *Boids*):

1. **Cohesion rule**: Move towards the center (median point) of the individuals in your neighborhood
2. **Separation rule**: Move away if you get too close to an individual
3. **Alignment rule**: Move in approximately the same direction as your neighbors

To implement this, you use *JGameGrid* again in order to keep the effort of the animation low. It helps to use a grid with pixel sizes and to specify the position, velocity, and acceleration of the actors with float vectors from the class *GGVector*. In each simulation period you first determine the new acceleration vector according to the three rules using *setAcceleration()*. This results in the new velocity and position vectors

$$\vec{v}' = \vec{v} + \vec{a} * dt \qquad \text{and} \qquad \vec{r}' = \vec{r} + \vec{v} * dt$$

Since the absolute time scale is insignificant, you can set the time increment to *dt = 1*.

Applying the separation rule not only leads to a rejection between the closely flying birds, but also between them and obstacles (in this case, trees).

You have to particularly deal with the edge of the flying area (the wall). For this, there are various possibilities to choose from. You could, for example, use a toroidal topology, where the birds leaving the area on one side enter it again on the other. In this case, the birds are simply forced to turn around at the edge.

```
from gamegrid import *
import math
import random

# ==================== class Tree ======================
class Tree(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/tree1.png")

# ==================== class Bird ======================
class Bird(Actor):
    def __init__(self):
        Actor.__init__(self, True, "sprites/arrow1.png")
        self.r = GGVector(0, 0)  # Position
        self.v = GGVector(0, 0)  # Velocity
        self.a = GGVector(0, 0)  # Acceleration

    # Called when actor is added to gamegrid
    def reset(self):
        self.r.x = self.getX()
        self.r.y = self.getY()
        self.v.x = startVelocity * math.cos(math.radians(self.getDirection()))
        self.v.y = startVelocity * math.sin(math.radians(self.getDirection()))

    # ----------- cohesion ---------------
    def cohesion(self, distance):
        return self.getCenterOfMass(distance).sub(self.r)

    # ----------- alignment --------------
    def alignment(self, distance):
        align = self.getAverageVelocity(distance)
        align = align.sub(self.v)
        return align

    # ----------- separation -------------
    def separation(self, distance):
        repulse = GGVector()
        # ------ from birds ------
        for p in birdPositions:
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance and d != 0:
                repulse = repulse.add(dist.mult((d - distance) / d))

        # ------ from trees ------
        trees = self.gameGrid.getActors(Tree)
        for actor in trees:
            p = GGVector(actor.getX(), actor.getY())
```

```
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance and d != 0:
               repulse = repulse.add(dist.mult((d - distance) / d))
        return repulse

    # ----------- wall interaction -------
    def wallInteraction(self):
        width = self.gameGrid.getWidth()
        height = self.gameGrid.getHeight()
        acc = GGVector()
        if self.r.x < wallDist:
            distFactor = (wallDist - self.r.x) / wallDist
            acc = GGVector(wallWeight * distFactor, 0)
        if width - self.r.x < wallDist:
            distFactor = ((width - self.r.x) - wallDist) / wallDist
            acc = GGVector(wallWeight * distFactor, 0)
        if self.r.y < wallDist:
            distFactor = (wallDist - self.r.y) / wallDist
            acc = GGVector(0, wallWeight * distFactor)
        if height - self.r.y < wallDist:
            distFactor = ((height - self.r.y) - wallDist) / wallDist
            acc = GGVector(0, wallWeight * distFactor)
        return acc

    def getPosition(self):
        return self.r

    def getVelocity(self):
        return self.v

    def getCenterOfMass(self, distance):
        center = GGVector()
        sum = 0
        for p in birdPositions:
            dist = p.sub(self.r)
            d = dist.magnitude()
            if d < distance:
                center = center.add(p)
                sum += 1
        if sum != 0:
            return center.mult(1.0/sum)
        else:
            return center

    def getAverageVelocity(self, distance):
        avg = GGVector()
        sum = 0
        for i in range(len(birdPositions)):
            p = birdPositions[i]
            if (self.r.x - p.x) * (self.r.x - p.x) + \
               (self.r.y - p.y) * (self.r.y - p.y) < distance * distance:
                avg = avg.add(birdVelocities[i]);
                sum += 1
        return avg.mult(1.0/sum)

    def limitSpeed(self):
        m = self.v.magnitude()
        if m < minSpeed:
            self.v = self.v.mult(minSpeed / m)
        if m > maxSpeed:
            self.v = self.v.mult(minSpeed / m)

    def setAcceleration(self):
        self.a = self.cohesion(cohesionDist).mult(cohesionWeight)
        self.a = self.a.add(self.separation(separationDist).mult(separationWeight))
        self.a = self.a.add(self.alignment(alignmentDist).mult(alignmentWeight))
        self.a = self.a.add(self.wallInteraction())
```

```
    def act(self):
        self.setAcceleration()
        self.v = self.v.add(self.a) # new velocity
        self.limitSpeed()
        self.r = self.r.add(self.v) # new position
        self.setDirection(int(math.degrees(self.v.getDirection())))
        self.setLocation(Location(int(self.r.x), int(self.r.y)))


# ================== global section ==================
def populateTrees(number):
    blockSize = 70
    treesPerBlock = 10
    for block in range(number // treesPerBlock):
        x = getRandomNumber(800 // blockSize) * blockSize
        y = getRandomNumber(600 // blockSize) * blockSize
        for t in range(treesPerBlock):
            dx = getRandomNumber(blockSize)
            dy = getRandomNumber(blockSize)
            addActor(Tree(), Location(x + dx, y + dy))

def generateBirds(number):
    for i in range(number):
        addActorNoRefresh(Bird(), getRandomLocation(),
                                  getRandomDirection())
    onAct()  # Initialize birdPositions, birdVelocities

def onAct():
    global birdPositions, birdVelocities
    # Update bird positions and velocities
    birdPositions = []
    birdVelocities = []
    for b in getActors(Bird):
        birdPositions.append(b.getPosition())
        birdVelocities.append(b.getVelocity())

def getRandomNumber(limit):
    return random.randint(0, limit-1)

# coupling constants
cohesionDist = 100
cohesionWeight = 0.01
alignmentDist = 30
alignmentWeight = 1
separationDist = 30
separationWeight = 0.2
wallDist = 20
wallWeight = 2
maxSpeed = 20
minSpeed = 10
startVelocity = 10
numberTrees = 100
numberBirds = 50

birdPositions  = []
birdVelocities  = []

makeGameGrid(800, 600, 1, False)
registerAct(onAct)
setSimulationPeriod(10)
setBgColor(makeColor(25, 121, 212))
setTitle("Swarm Simulation")
show()
populateTrees(numberTrees)
generateBirds(numberBirds)
doRun()
```

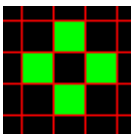
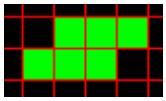
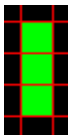
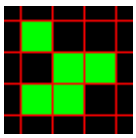
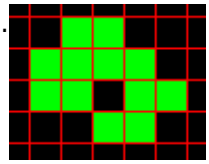
Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The simulation is dependent on several **coupling constants** that determine the "strength" of the interaction. Their values are very sensitive and you may eventually need to adjust them to the performance of your computer. Again, the callback *onAct()* is activated using *registerAct()* so that it is automatically called in every simulation cycle. The birds are moved with the method *act()* of the class *bird*.

EXERCISES

1. Study the behavior of the following patterns in Game of Life:

a.  b.  c.  d.  e. 

2. Describe three typical swarm behaviors that occur in the animal world. For each example, think about why the animals join together in a swarm.

3*. In the swarm simulation, introduce three raptors who follow a flock of birds that they are avoided by. Instructions: For the raptors, use the sprite image *arrow2.png*.

4*. Make it so that the raptors from exercise 3 eat the flock birds at a collision.

ADDITIONAL MATERIAL: LIST COMPREHENSION

Lists can be created neatly in *Python* with a special notation. It is based on mathematical notation from set theory

Mathematics	*Python*
S = {x : x in {1...10}}	s = [x for x in range(1, 11)]
T = {x^2 : x in {0...10}}	t = [x**2 for x in range(11)]
V = {x | x in S und x gerade}	v = [x for x in s if x % 2 == 0]
$M = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	m = [[0 for x in range(3)] for y in range(3)]

```
>>> s = [x for x in range(1, 11)]
>>> s
 < [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> t = [x**2 for x in range(11)]
 < [0, 1, 4, 9, 16, 25, 35, ..., 100]
>>> v = [x for x in s iff x% 2 == 0]
 < [2, 4, 6, 8, 10]
>>> m = [[0 for x in range(3)] for y in range(3)]
>>> m
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

8.10 RANDOM WALK

INTRODUCTION

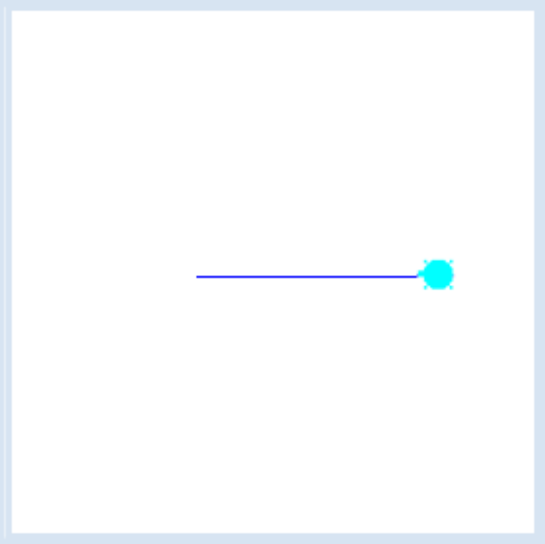
Many phenomena are determined by chance in daily life and you often have to make decisions based on probabilities. For example, you may decide whether you should select one mode of transport over the other based on their probability of being involved in an accident. In such situations, the computer can be an important tool with which you can examine potential dangers using simulations, without actually risking anything.

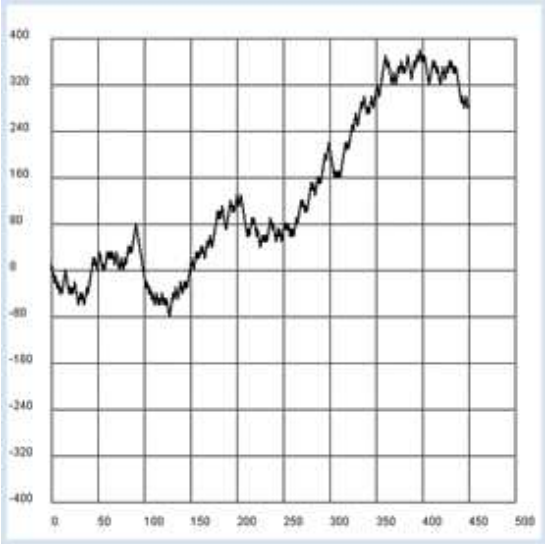
In the following, you assume that you got lost and can no longer find your way home. Thereby you observe a movement, called Random Walk, that happens at single time intervals where steps of equal length are chosen in a random direction. Even though such a movement does not necessarily correspond to reality, you get to know important characteristics that can be later applied to real systems, for instance the modeling of the stock market in financial mathematics, or the displacement of molecules.

PROGRAMMING CONCEPTS: *Random walk, Brownian motion*

ONE-DIMENSIONAL RANDOM WALK

Once again, turtle graphics is helpful in representing the simulation graphically. At time 0, the turtle is located at the position x = 0 and move along the x-axis at steps of equal length. At each step of time, the turtle "decides" whether to make a step to the left or to the right. In your program, it decides on one of the two options with the same probability p = ½ (symmetric random walk).






```
from gturtle import *
from gpanel import *
import random

makeTurtle()
makeGPanel(-50, 550, -480, 480)
windowPosition(880, 10)
drawGrid(0, 500, -400, 400)
```

```
title("Mean distance versus time")
lineWidth(2)
setTitle("Random Walk")

t = 0
while t < 500:
    if random.randint(0, 1) == 1:
        setHeading(90)
    else:
        setHeading(-90)
    forward(10)
    x = getX()
    draw(t, x)
    t += 1
print "All done"
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

You are probably surprised that, in most cases, the turtle gradually moves away from the starting point even though it makes every step, either a left or a right one, with the same likelihood. In order to investigate this important result more closely, your next program will determine the average distance *d* from the starting point after *t* steps in 1,000 walks.

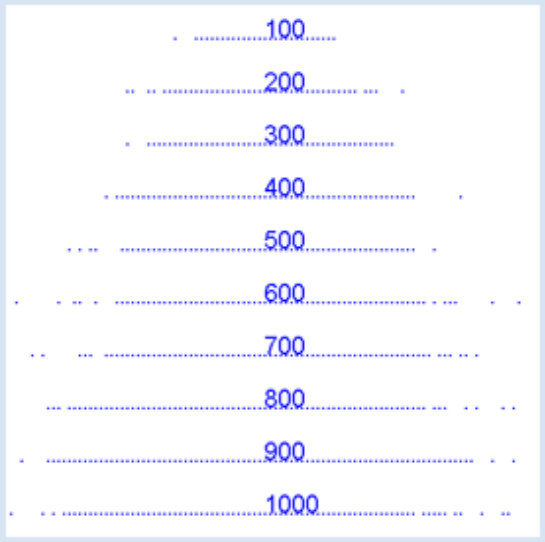
THE SQUARE ROOT OF TIME RULE

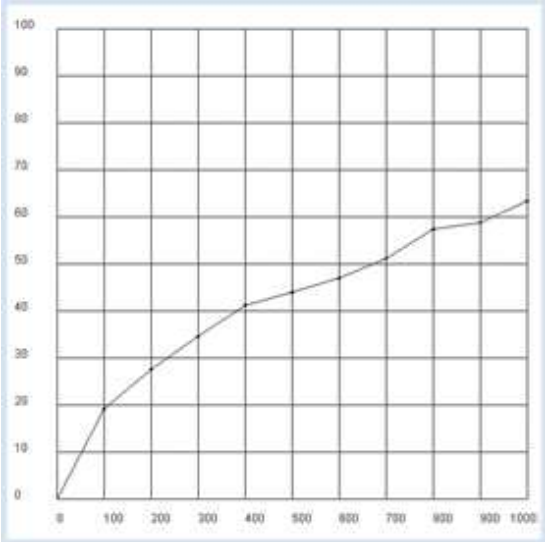
Repeat the simulation 1,000 times with a certain number of steps *t* at the fixed y position and draw the end position as a point. In order to speed up the results, you can hide the turtle and avoid drawing any traces. With every simulation experiment, you determine a distance *r* of the ending point from the starting point. For each set of experiments with a certain *t*, you will get an average distance *d* of the distances *r*, which you mark in a GPanel graphic.

```
from gturtle import *
from gpanel import *
import math
import random

makeTurtle()
makeGPanel(-100, 1100, -10, 110)
windowPosition(850, 10)
drawGrid(0, 1000, 0, 100)
title("Mean distance versus time")
ht()
pu()

for t in range(100, 1100, 100):
    setY(250 - t / 2)
    label(str(t))
    sum = 0
    repeat 1000:
        repeat t:
           if random.randint(0, 1) == 1:
               setHeading(90)
           else:
               setHeading(-90)
           forward(2.5)
        dot(3)
        r = abs(getX())
        sum += r
        setX(0)
    d = sum / 1000
```

```
    print "t =", t, "d =", d, "q = d / sqrt(t) =", d / math.sqrt(t)
    draw(t, d)
    fillCircle(5)
print "all done"
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)





MEMO

As you can see in the graphic visualization, the average value of the distance from the starting point grows with an increasing step size and time *t*. In the console, you also calculate the quotient *q = d / sqrt(t)*. Because it is almost constant, the assumption suggests that *d = q * sqrt(t)* holds exactly, or:

The average distance to the starting point increases by the square root of time.

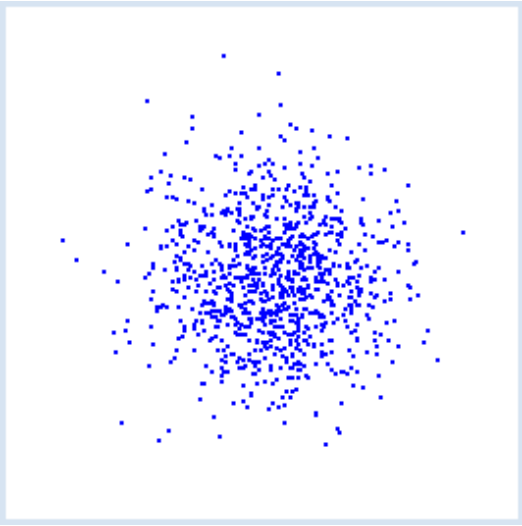
DRUNKEN MAN'S WALK

It gets even more exciting once the turtle can move in two directions. This is called a two-dimensional random walk. The turtle always still makes the same length of steps, but each step is taken towards a random direction.

Mathematicians and physicists often wrap the problem into the following story, which is not to be taken too seriously.

"A drunk person tries to return home after making a pub tour. Since they have lost their orientation, they always makes one step in a random direction. How far away have they moved from the pub (on average)?"

You only need to slightly change your previous program. You again examine how the average of the distance depends on time.



```
from gturtle import *
from gpanel import *
```

```
import math

makeTurtle()
makeGPanel(-100, 1100, -10, 110)
windowPosition(850, 10)
drawGrid(0, 1000, 0, 100)
title("Mean distance versus time")
ht()
pu()

for t in range(100, 1100, 100):
    sum = 0
    clean()
    repeat 1000:
        repeat t:
           fd(2.5)
           setRandomHeading()
        dot(3)
        r = math.sqrt(getX() * getX() + getY() * getY())
        sum += r
        home()
    d = sum / 1000
    print "t =", t, "d =", d, "q = d / sqrt(t) =", d / math.sqrt(t)
    draw(t, d)
    fillCircle(5)
    delay(2000)
print "all done"
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

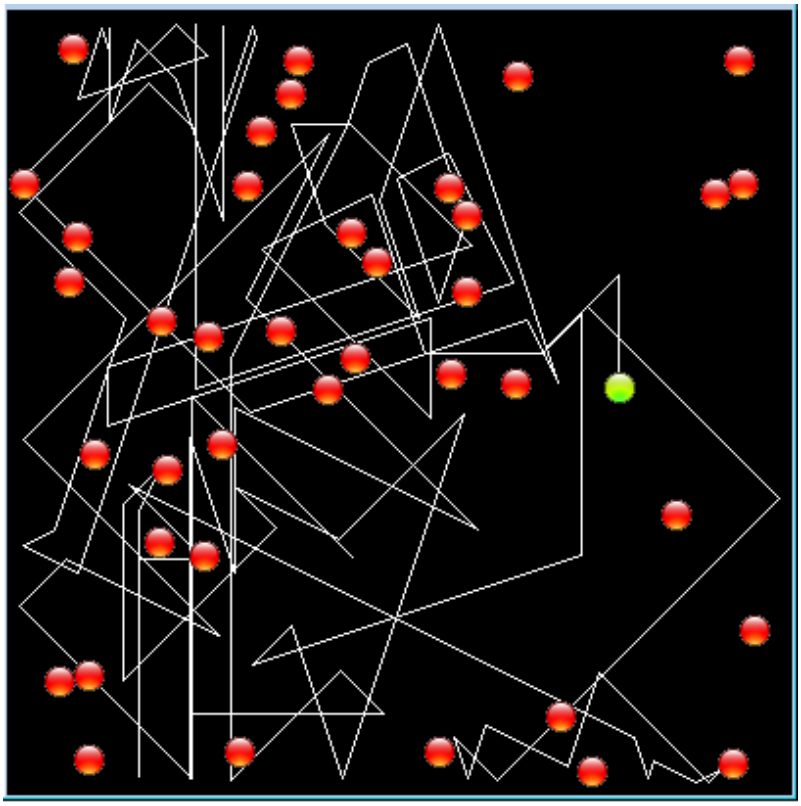
MEMO

The square root of time rule also applies for two-dimensional situations. In 1905, no less a person than Albert Einstein proved this rule for gas particles in his famous essay "On the Movement of Small Particles Suspended in Stationary Liquids Required by the Molecular-Kinetic Theory of Heat" and thus,provided us with a theoretical explanation of the Brownian motion. One year later, M. Smoluchowski had a different idea but came to the same results.

BROWNIAN MOVEMENT

As early as 1827, the biologist Robert Brown had observed through a microscope that pollen grains in a drop of water make irregular twitching movements. He suspected an inherent vigor in the pollen. It was not until the discovery of the molecular structure of matter, that it became clear the thermal motion of water molecules colliding with the pollen are responsible for the phenomenon.

Brownian motion can be nicely demonstrated in a computer simulation where the molecules are modeled as small spheres that swap velocities in collisions. The simulation can be easily implemented using *JGameGrid* because you can regard a particle collision as an event. To do this, you derive the class *CollisionListener* from *GGActorCollisionListener* and implement the callback *collide()*. You add each particle to the listener using *addActorCollisionListener()*. You may set the type and size of the collision area with *setCollisionCircle()*. For simplicity the 40 particles are arranged into 4 groups of velocities.



```
from gamegrid import *

# ==================== class Particle ====================
class Particle(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/ball.gif", 2)

    # Called when actor is added to gamegrid
    def reset(self):
        self.oldPt = self.gameGrid.toPoint(self.getLocationStart())

    def advance(self, distance):
        pt = self.gameGrid.toPoint(self.getNextMoveLocation())
        dir = self.getDirection()
        # Left/right wall
        if pt.x < 5 or pt.x > w - 5:
            self.setDirection(180 - dir)
        # Top/bottom wall
        if pt.y < 5 or pt.y > h - 5:
            self.setDirection(360 - dir)
        self.move(distance)

    def act(self):
        self.advance(3)
        if self.getIdVisible() == 1:
            pt = self.gameGrid.toPoint(self.getLocation())
            self.getBackground().drawLine(self.oldPt.x, self.oldPt.y, pt.x, pt.y)
            self.oldPt.x = pt.x
            self.oldPt.y = pt.y

# ==================== class CollisionListener =========
class CollisionListener(GGActorCollisionListener):
   # Collision callback: just exchange direction and speed
    def collide(self, a,  b):
        dir1 = a.getDirection()
        dir2 = b.getDirection()
        sd1 = a.getSlowDown()
        sd2 = b.getSlowDown()
        a.setDirection(dir2)
        a.setSlowDown(sd2)
```

```
        b.setDirection(dir1)
        b.setSlowDown(sd1)
        return 10  # Wait a moment until collision is rearmed

# ================== Global section ====================
def init():
    collisionListener = CollisionListener()
    for i in range(nbParticles):
        particles[i] = Particle()
        # Put them at random locations, but apart of each other
        ok = False
        while not ok:
            ok = True
            loc = getRandomLocation()

        for k in range(i):
            dx = particles[k].getLocation().x - loc.x
            dy = particles[k].getLocation().y - loc.y
            if dx * dx + dy * dy < 300:
                ok = False
        addActor(particles[i], loc, getRandomDirection())
        # Select collision area
        particles[i].setCollisionCircle(Point(0, 0), 8)
        # Select collision listener
        particles[i].addActorCollisionListener(collisionListener)
        # Set speed in groups of 10
        if i < 10:
            particles[i].setSlowDown(2)
        elif i < 20:
            particles[i].setSlowDown(3)
        elif i < 30:
            particles[i].setSlowDown(4)
    #  Define collision partners
    for i in range(nbParticles):
        for k in range(i + 1, nbParticles):
            particles[i].addCollisionActor(particles[k])
    particles[0].show(1)

w = 400
h = 400
nbParticles = 40
particles = [0] * nbParticles

makeGameGrid(w, h, 1, False)
setSimulationPeriod(10)
setTitle("Brownian Movement")
show()
init()
doRun()
```

Highlight program code (Ctrl+C to copy, Ctrl+V to paste)

MEMO

The molecules are modeled in the class Particle which is derived from Actor. They have two sprite images, one red and one green for a specially distinguished molecule whose path you are tracking. The green image corresponds to the spriteID = 1 which you test in act() to draw the trail with drawLine().

EXERCISES

1. A person moves from x = 0 with the same probability p = ½ to the right and q = ½ to the left in a one-dimensional Random walk with 100 steps. Execute the simulation 10,000 times and determine the frequency distribution of the end position. Display it in a *GPanel*.

2. As you can see in exercise 1, the probability of finding yourself at the starting point again after 100 steps is the highest. How can this be in agreement with the square root of time rule?

3. Conduct the same simulation, but with the probability p = 0.8 of a right step and q = 0.2 for a left step. ist.

4*. With a one-dimensional random walk, the theory provides a binomial distribution. The probability that you find yourself at the coordinate *x* after *n* steps amounts to the following, for an even *n* and *x*:

$$P(x, n) = \frac{n!}{((n + x)/2)!((n - x)/2)!} \quad p^{(n+x)/2} q^{(n-x)/2}$$

Plot the theoretical course as a curve in the histogram of exercise 1.

chapter nine

DATABASES & SQL

Learning Objectives

- You can explain why files are important in computer science.
- You know how to read, convert, and save data from a text file.
- You know some important features of online databases and can install your own database server.
- You are aware that nowadays a huge amount of information is stored electronically in online databases and social networks.
- You can generate tables using SQL on a database server and create, read, and modify data sets.
- You are on track to being able to set up and manage a simple online reservation system.
- You know how catching exceptions works and why it is an important principle.

> Anyone who processes personal data must make certain that it is correct. He must take all reasonable measures to ensure that data that is incorrect or incomplete in view of the purpose of its collection is either corrected or destroyed. Any data subject may request that incorrect data be corrected.
>
> Switzerland. Federal Law on Data Protection, Article 5

9.1 PERSISTENCE, FILES

INTRODUCTION

Computer-stored information, called data, plays a central role in today's high-tech society. Although they are comparable to written text, there are several important differences:

- Data can only be read, saved, and processed with a computer system.
- Data are always coded as 0.1 values. They only receive information content and make sense when they are correctly interpreted (decoded).
- Data possess a certain life span. Temporary data exist as local variables for a short time in a certain program block or as global variables for the entire duration of the program. Persistent data, however, survive the duration of the program and can later be retrieved.
- Data have a visibility (availability). While certain data, such as personal data, can be read by anyone in a social network, there are also private data or other data that can be found on storage devices that are not generally accessible to the public.
- Data can be protected. Protection can be achieved by encryption or restrictions on access (access and password protection).
- Data can easily be transported on digital communication channels.

Persistent data can be written or read in the form of files with computer programs on physical storage devices (common are: Hard Drive (HD), Solid State Disk (SSD), memory card or USB stick).

Files consist of storage areas with a certain structure and a specific file format. Since the transfer of data, even over large distances, has become fast and cheap, files are stored on distant media (**clouds**) more and more frequently

Files are managed on the computer in a hierarchical directory structure, i.e. a specific directory can hold not only files, but also sub-directories. Files are accessible through their **file path** (short 'path') which contains the names of the directories and the file. However, the file system is dependent on the operating system, and therefore there are certain differences between Windows, Mac, and Linux systems.

PROGRAMMING CONCEPTS: *Encoding, life span, visibility of data, file*

READING AND WRITING TEXT FILES

You have already learned how to read text files in the chapter *Internet*. In text files, characters are stored sequentially, but one can get a line structure similar to that of a piece of paper by inserting end of line symbols. It is exactly here that the operating systems differ: While Mac and Linux use the ASCII characters <line feed> as an end of line (EOL), Windows uses the combination of <carriage return><line feed>. In *Python* these characters are coded with \r and \n [**more...**].

You will use an English dictionary in your program, which is available as a text file. You can download it (*tfwordlist.zip*) from **here** and unzip it in any directory. Copy the file *words-1$.txt* in the directory where your program is located.

You will now take a look at the interesting question of which words are palindromes. A palindrome is a word that reads the same forward or backward, without considering the case of the letters.

With *open()* you receive a file object *f* that provides you with access to the file. Later you can run

through all the data with a simple *for* loop. Thereby you should pay attention to the fact that each line contains an end of line symbol that must first be cut off with a slice operation before you read the word backwards. In addition, you should convert all characters to lowercase using *lower()*.

Reversing a string is somewhat tricky in *Python,* since the slice operation also allows for negative indices, and in this case the indices counting begins at the end of the string. If you select a step parameter -1, the string is run through backwards.

```
def isPalindrom(a):
    return a == a[::-1]

f = open("worte-1$.txt")

print "Searching for palindroms..."
for word in f:
    word = word[:-1] # remove trailing \n
    word = word.lower() # make lowercase
    if isPalindrom(word):
        print word
f.close()
print "All done"
```

With the method *readline()* you can also read line by line. You can imagine a line pointer to be advanced at each call. Once you have made it to the end of the file, the method returns an empty string. Save the result in a file named *palindrom.txt*. In order to write to the file, you must first create it with *open()* passing it the parameter "w" (for write). Then, you can write to it using the method *write()*. Do not forget to use the method *close()* at the end, so that all the characters are for sure written to the file and the operating system resources are released again.

```
def isPalindrom(a):
    return a == a[::-1]

fInp = open("worte-1$.txt")
fOut = open("palindrom.txt", "w")

print "Searching for palindroms..."
while True:
    word = fInp.readline()
    if word == "":
        break
    word = word[:-1] # remove trailing \n
    word = word.lower() # make lowercase
    if isPalindrom(word):
        print word
        fOut.write(word + "\n")
fInp.close()
fOut.close()
print "All done"
```

MEMO

When you open text files using *open(path, mode)* the user mode is specified with the parameter *mode*.

Mode	Description	Comment
"r" (read)	Read only	File must already exist. Parameter can be omitted
"w" (write)	Create and write file	An existing file is deleted first

"a" (append)	Attach at the end of the file	Create the file if it does not already exist
"r+"	Read and attach	File must already exist

Once you have read all the lines of a file and want to read it again, you have to either close the file and reopen it or simply call the method *seek(0)* of the file object. You can also read the entire contents of the text file in a string using

```
text = f.read()
```

and then close the file. You can create a list with the line strings (without end of line) using

```
textList = text.splitlines()
```

Other important file operations:

import os os.path.isfile(path)	Returns True, if the file exists
import os os.remove(path)	Deletes a file

SAVING AND RETRIEVING OPTIONS OR GAME FILES

Files are often used to save information so that it can be retrieved again during the next execution of the program, for example program settings (options) which are made by the user to customize their program. Maybe you would also like to save the current (game) state of a game so that you are able to continue playing in exactly the same situation.

Options and states usually save nicely as key-value pairs, where the key is an identifier for the value. For example, there are certain configuration values (setup parameters) for the *TigerJython* IDE:

Key	**Value**
"autosave"	True
"language"	"de"

As you learned in chapter 6.3, you can save such key-value pairs in a Python *dictionary* that you can very easily save and retrieve as (binary) files with the module *pickle*. In the following example, you save the current position and direction of the lobsters and also the position of the simulation cycle regulator before closing the game window. The saved values will then be restored at the next start.


```
import pickle
import os
from gamegrid import *

class Lobster(Actor):
   def __init__(self):
      Actor.__init__(self, True, "sprites/lobster.gif");

   def act(self):
      self.move()
```

```
        if not self.isMoveValid():
            self.turn(90)
            self.move()
            self.turn(90)

makeGameGrid(10, 2, 60, Color.red)
addStatusBar(30)
show()

path = "lobstergame.dat"
simulationPeriod = 500
startLocation = Location(0, 0)
if os.path.isfile(path):
    inp = open(path, "rb")
    dataDict = pickle.load(inp)
    inp.close()
    # Reading old game state
    simulationPeriod = dataDict["SimulationPeriod"]
    loc = dataDict["EndLocation"]
    location = Location(loc[0], loc[1])
    direction = dataDict["EndDirection"]
    setStatusText("Game state restored.")
else:
    location = startLocation
    direction = 0

clark = Lobster()
addActor(clark, startLocation)
clark.setLocation(location)
clark.setDirection(direction)
setSimulationPeriod(simulationPeriod)

while not isDisposed():
    delay(100)
gameData = {"SimulationPeriod": getSimulationPeriod(),
            "EndLocation": [clark.getX(), clark.getY()],
            "EndDirection": clark.getDirection()}
out = open(path, "wb")
pickle.dump(gameData, out)
out.close()
print "Game state saved"
```

MEMO

A dictionary can be saved in a file with the method **pickle.dump()**. It will be a binary file that you are not able to edit directly.

EXERCISES

1. Search for anagrams in the file *words-1$.txt* (Anagrams are two words with the same letters, but in different order. You can ignore the case of the letters). Write the anagrams that you found to a file *anagram.txt.*

2. The text below was encrypted by anagramming, i.e. the original words were substituted by those with permuted letters.
   IINFHS SOLOHC AEMK SIWREKOFR

   Try to decrypt the text with help from the word lists (*words-1$.txt)* [**more...**].

3. Create your own ciphertext that can be unambiguously decrypted with the word lists.

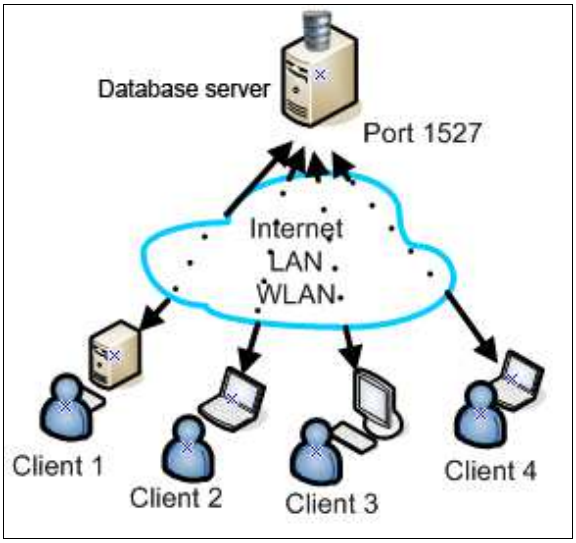
9.2 ONLINE DATABASES

INTRODUCTION

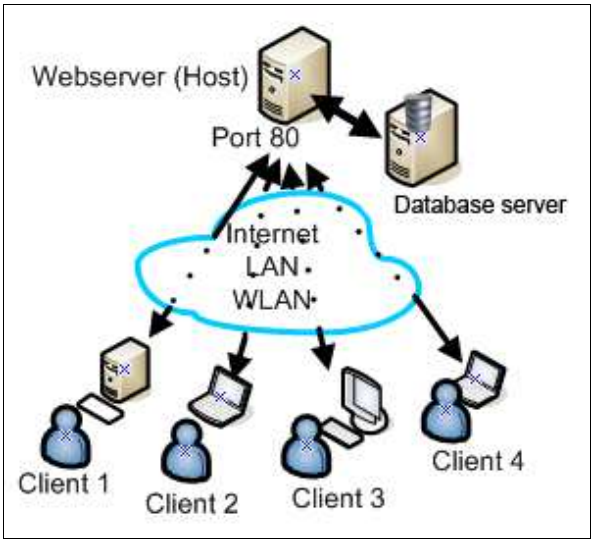
Databases are exceptionally important in today's world. Their main purpose is to store information in a structured manner that one can easily retrieve it with search and link criteria. Due to the interconnectedness of the Internet and the widespread use of social networks, there is a gigantic amount of information stored at millions of database hosts. Therefore, it is important that you learn how data is handled in a database. This will also help you to better assess the risks associated with the use of database-based systems.

In most computerized databases, information is saved in the form of tables which are interrelated. They are thus called **relational databases**. In order to deal with the tables efficiently, a program bundle called the relational database management system (RDBMS) must be available. Databases should therefore be understood as the collection of data that they contain, together with the corresponding management tools.

Simple databases can be located locally on a PC and operated by a single person, for example for the management of CDs or books. But most databases are hosted on the Internet and are therefore accessible to many users simultanously (online databases). These client-server systems include a computer that acts as database server (also called a host) and multiple distributed clients. The data communication works similarly to a web server using the TCP protocol over a TCP port (e.g. 3306 for MySQL or 1527 for Derby). A specific application program that is written in any high-level programming language such as *Python* runs on the client and connects to the host.



Direct connection



Indirect connection

Often the client is not directly connected to the database server, but rather indirectly through a web server. The client then only runs one of the known web browsers. In this case, the specific program for the exchange of data with the database servers is located on the web server and must also be written in a suitable programming language (often PHP). The procedures are similar in both cases and demand good programming knowledge in terms of databases, which you will acquire here [**more...**].

PROGRAMMING CONCEPTS: *Database, table, database server, exceptions with try-except*

YOUR OWN DATABASE SERVER

Access to existing database servers on the Internet is subject to strong security restrictions since you want to avoid unauthorized data from being saved or changed. For these reasons, you install your own database server on your PC that you can either use directly from there, or from other clients within a LAN/WLAN connection. You usually have to authenticate a database with a username and password in order to access it [**more...**].

In the following example you will instal and use Derby, a free product of the Apache organization developed by the world famous Apache web server. (You could also use any other database software, e.g. MySQL.) To install Derby, follow the instructions listed below

1. Download the file *tjderby.zip* from **here.**
2. Unpack it and copy the files into the subdirectory *Lib of the directory where* tigerjython2.jar is located.
3. Go into the directory *Lib* with a command shell and start the server with
   `java -jar derbynet.jar start`
   Heads up! You have to have administrator privileges [**more...**].

As an alternative you can start the server by using the files *startderby.bat* (for Windows) or *startderby* (for Linux/Mac) located in the directory *Lib*. Create a link for these files so that they are just a click away. The server is only visible on the local PC (localhost). To open it to the outside, you need to start it by setting the IP address of your PC. If your IP addres is 10.1.1.123, start it with

```
java -jar derbynet.jar start -h 10.1.1.123
```

or with

```
startderby 10.1.1.123.
```

The database server can manage several databases simultaneously. You can gain access to a database with the database name and a username and password. You need to connect to the server using *getDerbyConnection()* in order to create and use a database. If the database does not already exist, it will then be created.

Since you learn how to create a reservation system for the concert hall at the fictitious concert venue Casino, select *casino* as a database name and *admin/solar* as the username and password combination.

After the successful establishment of a connection, you receive a connection object *con* from *getDerbyConnection()* with which you can request an access key (*cursor)* to the database *by* calling the method *cursor()*. Finally, all doors to the database *casino* are open to you.
You can consider the database as an object to which you can send commands and which executes them accordingly. The commands are written in a somewhat standardized database language SQL (Structured Query Language) that leans heavily on English slang, so that pretty much anyone can figure it out. You pack a SQL command into a string and execute it with the method *execute(SQL)*. To make it a bit more clear, you write the SQL language elements in capital letters. However, it also works with lower case letters or a mix between the two.

As a first step, create a database table with the SQL command *CREATE TABLE*. As usual, the table consists of rows and columns. The columns, also called fields, specify what information you want to store in the table. You need to give the field a name and a data type for this. The most important types are listed in the following table:

SQL data type (for Derby)	**Description**
VARCHAR	String, appropriate length (<=255)
CHAR(M)	String, M characters, fixed length M <= 255
INTEGER	Whole number (4 bytes integer)

FLOAT	Decimal number (64 bit double precision)
DATE	Date (java.sql.date)

To administer the seat reservations in the casino hall for a certain concert, you add a date in the format yyyymmdd (y: year, m: month, d: day) to the table name *res*. As other fields you choose seat number *seat*, *booked* with the letters N or Y, and a customer number *cust* that identifies the person who reserved a seat.

```
#Db2a.py

from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
SQL = "CREATE TABLE res_20140115 (seat INTEGER,booked CHAR(1),cust INTEGER)"
cursor.execute(SQL)
con.commit()
cursor.close()
con.close()
print "Table created"
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

If the program finishes without problems, you will know that you set up the database server successfully. If you get an error message carefully read through the chapter once more and try to repeat each step.

With databases, it is typical that SQL commands only affect the database after *commit()* is called. It is thus ensured that multi-state database transactions can be considered as entities that are always executed as a whole or upon an error message not at all. Finally, all resources have to be freed again with *close()*. You should carefully memorize "tidying up" with

```
con.commit()
cursor.close()
con.close()
```

because it must never be missing in the following.

If you run the program when the table already exists, it aborts with an error message. The bad thing about this is that all futher database connections fail until *TigerJython* is restarted.

CATCHING ERRORS WITH TRY-EXCEPT

If the table that you want to create already exists, calling *execute()* crashes the program. It can also be said that the program throws an exception. Later parts of the program will not be executed, especially not the important "cleaning up" part. As a result, specific resources are not freed which can block other programs or even the entire PC. Therefore, it is very important to catch such errors.

Catching a thrown exception in *Python* is very simple. You put the instructions that may cause an exception into a try-except block. If the exception occurs the program branches immediately into

the except part. After the execution of the except part, the program will continue with the statement immediately after the try-except block. Optionally, an else part can be specified that is executed if no exception occurs.

When executing SQL commands in the future, you thus write your program so that it intercepts any possible exceptions.

```
from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
   SQL = "CREATE TABLE res_20140115 (seat INTEGER,booked CHAR(1),cust INTEGER)"
   cursor.execute(SQL)
except Exception, e:
   print "SQL executing failed.", e
else:
   print "Table created"
con.commit()
cursor.close()
con.close()
```

MEMO

Critical parts of the program should be put in a try-except block so that the clean up still runs if the program is aborted. The except command can extract from the parameter e important information about the type of error..

INSERTING DATA INTO THE TABLE

In the next phase you will want to fill the table with initialization data, so label all seats as available and enter the customer number 0. To do this you use the SQL INSERT command, for example for the set with the number 1:

```
INSERT INTO res_20140115 VALUES (1, 'N', 0)
```

With this command you add a single row to the table. A single row of the table is also called a dataset or a record.

```
#Db2c.py

from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"
con = getDerbyConnection(serverURL, dbname, username, password)

cursor = con.cursor()
try:
    for seatNb in range(1, 31):
        SQL = "INSERT INTO " + table + " VALUES (" + str(seatNb) + ",'N',0)"
        cursor.execute(SQL)
```

```
except Exception, e:
    print "SQL executing failed. ", e
else:
    print "Table initialized"
con.commit()
cursor.close()
con.close()
```

MEMO

If you execute the program twice the record is inserted twice.

READING DATA FROM A TABLE

By now you are probably curious to see if the data are actually located in the table. In any case, today's software systems are so stable that you can assume that if there is no error message, the database transaction was indeed successful. In order to read a table, you should use the most famous SQL command:

```
SELECT * FROM res_20140115
```

By using an asterisk (*wildcard*) you are asking for all records to be read. After executing with *execute(),* you can retrieve the obtained records with the cursor method *fetchall()*. A list is returned in which the individual records are contained as *tuples* (an unalterable list). You can read the individual fields in it using indices.

The total number of records provided by the *SELECT* command is important. You get the number of records from the variable *rowcount*.

```
#Db2d.py

from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
    SQL = "SELECT * FROM " + table
    cursor.execute(SQL)
except Exception, e:
    print "SQL execution failed.", e
else:
    nbRecord = cursor.rowcount
    print "Number of records:", nbRecord
    result = cursor.fetchall() # list of tuples
    for record in result:
        print "seatNb:", record[0], " booked:", record[1]," cust:", record[2]
con.commit()
cursor.close()
con.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In addition to *fetchall()* you can also use *fetchmany(n)* and *fetchone()* to read the data. Each time you call one of the fetch methods, the cursor is, so to say, pushed forward as a pointer to the number of returned records (hence the name *cursor*) and the next call of *fetchall()*, *fetchmany()* or *fetchone()* delivers the records from the new position on. If the cursor reaches the end of the table this method returns the value *None*.

DELETING TABLES

You can finish the exercise by deleting the table that you just created. In the database language this is called a drop operation. The corresponding SQL command is:

```
DROP TABLE res_20140115
```

After this, your database *casino* no longer has any user-specified tables.

```
from dbapi import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
try:
    SQL = "DROP TABLE " + table
except Exception, e:
    print "SQL executing failed. ", e
else:
    print "Table removed"
con.commit()
cursor.close()
con.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

After deleting the table, your programs that read the data lead to an error message. You still have to adapt them for this special case.

The database named *pythondb* exists even after you delete the table. There are multiple files in the subdirectory *pythondb* of the directory where you copied the Derby software (in this case to *Lib/pythondb*). If you want to remove all traces of the exercise, you can simply delete the directory *pythondb*.

EXERCISES

1. Create a new reservation system where you can both make and cancel seat reservations. Start the database server *Derby* and run the programs Db2a.py and Db2c.py. This creates a new table and adds 30 records with empty seats.

   You can display all the records of your table with the following simplified program of Db2d.py

```
from dbapi import *

username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
table = "res_20140915"

def showAll():
    SQL = "SELECT * FROM " + table
    cursor.execute(SQL)
    con.commit()
    result = cursor.fetchall()
    for record in result:
        print "seatNb:", record[0]," booked:",record[1],"customer:",record[2]

con = getDerbyConnection(serverURL, dbname, username, password)
cursor = con.cursor()
showAll()
cursor.close()
con.close()
```

2. You would like to reserve seat 6 for customer number 33. You can do this by adding an update query to the above program before calling *showAll()*.

```
SQL = "UPDATE " + table + " SET booked='Y', cust=33 WHERE seat=6"
cursor.execute(SQL)
```

   Reserve seat number 5 for the same customer as well. Also reserve seats 10, 11, and 12 for customer number 34 and seats 17 and 18 for customer number 35.

3. You can display all reserved seats using the SQL query

```
SQL = "SELECT * FROM " + table + " WHERE booked='Y'"
cursor.execute(SQL)
```

   a. Change the function *showAll()* so that only the empty seats are shown
   b. Change it to display only the seat reservations of customer number 34.

4. Customer number 34 cancelled all of their reservations. Implement a corresponding update query.

5. After a renovation of the hall, seats 24 - 30 are no longer available. Delete these records from the table.

   To do this, you can use a delete query according to the following model

```
SQL = "DELETE * FROM " + table + " WHERE ..."
cursor.execute(SQL)
```

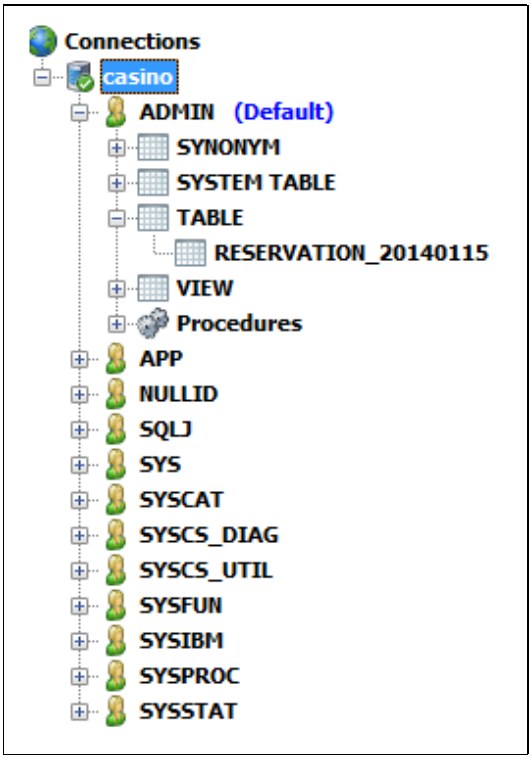
PROGRAMS FOR DATABASE MANAGEMENT

As a database administrator, you would like to manage your databases without actually having to program yourself. For this, there are various administrative tools available that you can find on the Internet. However, they are not always free. A distinction is made between command line tools and those with a graphical user interface. While professionals usually feel comfortable in a command shell, casual users prefer GUI programs.

A well-known administrator tool with a graphical user interface is *DBVisualizer*, and you can download a free version from here [**more...**]. You can use it to directly execute SQL commands and observe their effects. To install it, do the following:

1. Get the setup installer for the distribution customized for your platform from **http://www.dbvis.com/download**
2. Execute the downloaded file. Choose the default options.
3. Make sure that you have started the Derby database server, created a database *casino* with the table *res_20140115,* and initialized the records.
4. Start DBVisalizer and select *Tools* | *Connection Wizard*. In the dialog, select any one connection alias, e.g. *casino*. In the dialog Select Database Driver select *JavaDB/Derby Server*.
5. In the next dialog provide the user ID *admin*, password *sonar*, and database *casino* and then click on c*asino.*

   You will now see the following image in the navigation window and can open a window by double clicking on the table RES_2014015. You can display the contents of the table by clicking on the tab *Data*.



9.3 RESERVATION SYSTEM

INTRODUCTION

Nowadays, reserving seats on planes and at concert halls is done almost exclusively through online booking systems. The information is managed by online databases. Database specialists with great programming skills are responsible for the development of these systems. After studying the contents of this chapter, you can already rank among these specialists.

PROGRAMMING CONCEPTS: *Multi-user system, access conflict, sporadic error*

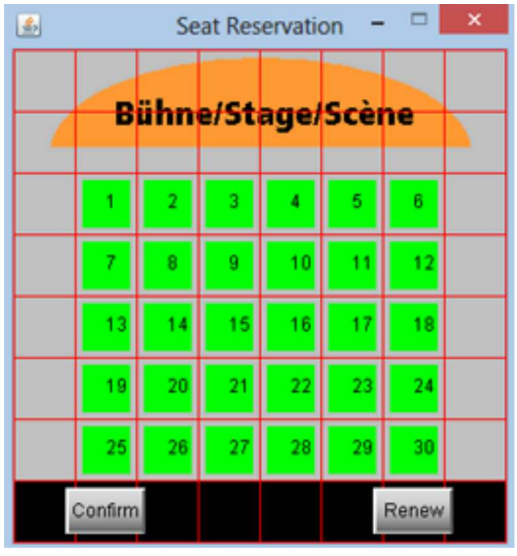
RESERVATION SYSTEM PART 1: USER INTERFACE

An attractive user interface is used to present the reservation system to the user. A user interface can be generated with a standalone program or as a web site. It is typical that the outline of the room or the aircraft is shown where the seat layout is displayed. Seats that are already reserved will be marked with a special color, for example the free seats could be green and the already reserved seats could be red.

The user can select a seating option by clicking on an empty seat. The seat then changes color, let's say to yellow. This selection is not yet transferred to the database server because often the user wants to reserve multiple seats at one time. The booking process is carried out only once they click the confirm button.

The game library *JGameGrid* is ideal for implementing the graphical user interface because the seats are usually arranged in a grid-like structure. The sprite identifier (0, 1, 2) can be used to save the current states (available, option, reserved).

In your example, you model a small concert or theater room with only 30 seats, numbered from 1 to 30. You embed them into the grid shown adjacently.



Use the class *GGButton* for the two buttons. In order to receive a notification of the button clicks, you need to define a separate button class *MyButton* that is derived from *GGButton* as from *GGButtonListener*. *The method buttonPressed() is called when a button is pressed.* You can find out which button was used using the parameter *button.*

You have already implemented a fully functional user interface with this short program. It is of course still missing a connecting to the database [**more...**] .

```
from gamegrid import *

def toLoc(seat):
    i = ((seat - 1) % 6) + 1
    k = ((seat - 1) // 6) + 2
    return Location(i, k)
```

```
def toSeatNb(loc):
    if loc.x < 1 or loc.x > 6 or loc.y < 2 or loc.y > 6:
       return None
    i = loc.x - 1
    k = loc.y - 2
    seatNb = k * 6 + i + 1
    return seatNb

class MyButton(GGButton, GGButtonListener):
    def __init__(self, imagePath):
        GGButton.__init__(self, imagePath)
        self.addButtonListener(self)

    def buttonClicked(self, button):
        if button == confirmBtn:
            confirm()
        if button == renewBtn:
            renew()

def renew():
    setStatusText("View refreshed")

def confirm():
    for seatNb in range(1, 31):
       if seats[seatNb - 1].getIdVisible() ==  1:
           seats[seatNb - 1].show(2)
           refresh()
    setStatusText("Reservation successful")

def pressCallback(e):
    loc = toLocation(e.getX(), e.getY())
    seatNb = toSeatNb(loc)
    if seatNb == None:
        return
    seatActor = seats[seatNb - 1]
    if seatActor.getIdVisible() == 0:  # free
        seatActor.show(1) # option
        refresh()
    elif seatActor.getIdVisible() == 1:  # option
       seatActor.show(0) # free
       refresh()


makeGameGrid(8, 8, 40, None, "sprites/stage.gif", False,
             mousePressed = pressCallback)
addStatusBar(30)
setTitle("Seat Reservation")
setStatusText("Please select free seats and press 'Confirm'")
confirmBtn = MyButton("sprites/btn_confirm.gif")
renewBtn = MyButton("sprites/btn_renew.gif")
addActor(confirmBtn, Location(1, 7))
addActor(renewBtn, Location(6, 7))
seats = []
for seatNb in range(1, 31):
    seatLoc = toLoc(seatNb)
    seatActor = Actor("sprites/seat.gif", 3)
    seats.append(seatActor)
    addActor(seatActor, seatLoc)
    addActor(TextActor(str(seatNb)), seatLoc)
show()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The conversion of seat numbers to locations and vice versa is done in two transformation functions. The names of such mapping functions are often prefixed with *to*.

RESERVATION SYSTEM PART 2: CONNECTING TO THE DATABASE

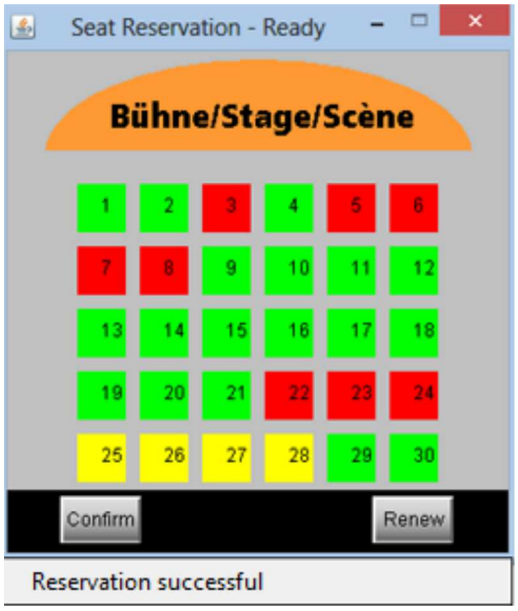
Online databases are multi-user systems. Because many clients manipulate the data at the same time, serious access conflicts can occur depending on the situation. It is in the nature of such problems to occur only sporadically, and they are therefore difficult to master. The following scenario describes a typical conflict case:

Customer A and Customer B make a reservation at the same time on a reservation system. At the beginning, A and B inquire about the current availability of the database just shortly one after the other. Both end up receiving the same information about the availability of seats (green = available, red = already reserved seats).

A and B can select the chosen seats as an option with a mouse click. The opted seats are then colored yellow. However, these options are not sent to the database since the customer might still want to change something or add more options. Only after some time, the faster determined customer A clicks on the confirm button to make their choice final. The database puts the seats in the state *reserved* (booked = 'Y'). However, customer B does not realize these changes, since the database cannot directly give feedback to customer B, and customer A is of course not in direct contact with B [**more...**]. After a certain amount of time, customer B also decides and presses the confirm button.

Just like Murphy said, "If anything can go wrong, it goes wrong" customers A and B have now chosen the same seats. What happens now? Are the seats assigned to customer B or will B's program even crash?

There is a simple solution to this access conflict: When a customer provides their options to the database, the database needs to be asked once again (just shortly before) if the seats are still actually free. If they are not, the database has no other choice than to politely notify the customer that the seats have already been assigned in the meantime [**more...**]



Note: The execution of this program requires you to restart the database server, create the table res_20140115, and the table must be filled with initialization data (see previous chapter).

```
from dbapi import *
from gamegrid import *

table = "res_20140115"
username = "admin"
password = "solar"
dbname = "casino"
serverURL = "localhost"
#serverURL = "10.1.1.123"

def toLoc(seat):
```

```
    i = ((seat - 1) % 6) + 1
    k = ((seat - 1) // 6) + 2
    return Location(i, k)

def toSeatNb(loc):
    if loc.x < 1 or loc.x > 6 or loc.y < 2 or loc.y > 6:
       return None
    i = loc.x - 1
    k = loc.y - 2
    seatNb = k * 6 + i + 1
    return seatNb

def pressCallback(e):
    if not isReady:
        return
    loc = toLocation(e.getX(), e.getY())
    seatNb = toSeatNb(loc)
    if seatNb == None:
        return
    seatActor = seats[seatNb - 1]
    if seatActor.getIdVisible() == 0:  # free
        seatActor.show(1) # option
        refresh()
    elif seatActor.getIdVisible() == 1:  # option
       seatActor.show(0) # free
       refresh()

class MyButton(GGButton, GGButtonListener):
    def __init__(self, imagePath):
        GGButton.__init__(self, imagePath)
        self.addButtonListener(self)

    def buttonClicked(self, button):
        if not isReady:
            return
        if button == confirmBtn:
            confirm()
        if button == renewBtn:
            renew()

def renew():
    global isReady
    try:
        SQL = "SELECT * FROM " + table
        cursor.execute(SQL)
        con.commit()
    except:
        setStatusText("Fatal error. Restart and try again.")
        isReady = False
        return

    result = cursor.fetchall()
    for record in result:
        seatNb = record[0]
        isBooked = (record[1] != 'N')
        if isBooked:
            seats[seatNb - 1].show(2)
        else:
            seats[seatNb - 1].show(0)
    refresh()
    setStatusText("View refreshed")

def confirm():
    global isReady
    try:
        # check if seats is still available
        for seatNb in range(1, 31):
           if seats[seatNb - 1].getIdVisible() == 1:
               SQL = "SELECT * FROM "  + table + "  WHERE seat=" + str(seatNb)
```

```
                cursor.execute(SQL)
                result = cursor.fetchall()
                for record in result:
                    if record[1] == 'Y':
                         setStatusText("One of the seats are already taken.")
                         return
        isReserved = False
        for seatNb in range(1, 31):
           if seats[seatNb - 1].getIdVisible() == 1:
                SQL = "UPDATE "  + table + " SET booked='Y' WHERE seat=" + \
                      str(seatNb)
                cursor.execute(SQL)
                isReserved = True
           con.commit()
        renew()
        if isReserved:
            setStatusText("Reservation successful")
        else:
            setStatusText("Nothing to do")
    except Exception, e:
        setStatusText("Fatal error. Restart and try again.")
        isReady = False

isReady = False
makeGameGrid(8, 8, 40, None, "sprites/stage.gif", False,
             mousePressed = pressCallback)
addStatusBar(30)
setTitle("Seat Reservation - Loading...")
confirmBtn = MyButton("sprites/btn_confirm.gif")
renewBtn = MyButton("sprites/btn_renew.gif")
addActor(confirmBtn, Location(1, 7))
addActor(renewBtn, Location(6, 7))
seats = []
for seatNb in range(1, 31):
    seatLoc = toLoc(seatNb)
    seatActor = Actor("sprites/seat.gif", 3)
    seats.append(seatActor)
    addActor(seatActor, seatLoc)
    addActor(TextActor(str(seatNb)), seatLoc)
show()

con = getDerbyConnection(serverURL, dbname, username, password)
if con == None:
    setStatusText("Fatal error. Connection to database failed")
else:
    cursor = con.cursor()
    renew()

    setTitle("Seat Reservation - Ready")
    setStatusText("Select free seats and press 'Confirm'")
    isReady = True
    while not isDisposed():
        delay(100)
    cursor.close()
    con.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In this exercise you also learn that it is not enough to simply write a program so that it returns the correct values only under ideal conditions and when operated correctly. Instead, it should be able to behave appropriately even under unfavorable conditions and manipulation errors. If an application has a proper user interface, the customer should be informed about the current state. They should also know if their actions were successful. Manipulations that are not allowed in a particular state have to be disabled.

A known way to disable actions is to use the flag *isReady* which determines whether key or mouse inputs are allowed. *isReady = False* is set at the beginning until the connection to the database is established (this may take some time for remote databases). Malfunctions are intercepted using a try-except block and set in the except part *isReady = False*.

You can test the reservation system with multiple user windows by repeatedly starting the *TigerJython* IDE on your computer and executing the same program.

EXERCISES

1. Expand the reservation system so that when you click on *Confirm,* the customer number is prompted (as an integer) and saved in the database. You can use the function *inputInt(prompt, False)* to read in, which does not end the program when you press the Cancel button, but rather returns *None*.

2. Write an administrator tool that displays the current assignment and a list of the reserved seats, as well as the customer number. In addition, it should be possible to undo a reservation by clicking on the red seats again. To make the list visible, open it in a console window with *console = console.init().* You can write into this list line by line using *console.println(text).* (*from ch.aplu.util import Console* required)

3. Write a tool with which you can create a table with customer numbers and the first and last names of the customer. Supplement the tool from exercise 2 so that the customer name also appears in the displayed list of the reserved seats.

SQL queries

SELECT * [column] **FROM** table [**WHERE** condition] [**ORDER BY** column [asc|desc]]

The options in square brackets are optional. Some examples:

SELECT * FROM tab	all records from the table tab
SELECT name, vorname FROM tab	only columns last name and first name
SELECT * FROM tab ORDER BY name	all records from the table tab sorted by name
SELECT * FROM tab WHERE anrede = 'Herr' ORDER BY name	all records with salutation “Mr.” sorted by name
SELECT * FROM tab WHERE name = 'Meier' and vorname = 'Luka'	searches for “Luka Meier” (both conditions (should be) met)
SELECT * FROM tab WHERE name = 'Meier' or name = 'Mayer'	one of the two conditions should be met?
SELECT * FROM tab WHERE name in ('Meier', 'Meyer', 'Müller')	name must be listed under the specified name(s)
SELECT * FROM tab WHERE name LIKE '%haus% '	all records that occur in the field name "house"
SELECT * FROM buch WHERE jahr between 1999 and 2014	numbers can be specified without quotes
SELECT count (*) FROM tab	determines the number of records in a table
SELECT concat (vorname, ' ', name) as vname FROM tab	connects last name and first name in a new field vname
SELECT sum(preis) FROM buch	determines the sum of all values in a table column

UPDATE table **SET** column1 = value1, [column2 = value2], [...] [**WHERE**condition]

UPDATE tab SET institut = 'PHBern'	replaces institute with PHBern in all record fields
UPDATE tab SET booked='Y', cust=33 WHERE seat=6	updates multiple columns
UPDATE tab SET anrede = 'Frau' WHERE anrede = 'f'	replaces f with Ms. in the field salutation
UPDATE buch SET preis = preis * 1.52	calculations can be performed in the update statement

DELETE FROM table **[WHERE** condition**]**

DELETE FROM tab	deletes all records in the table tab
DELETE FROM tab WHERE name = "Meier"	deletes the record with name = Meier

chapter ten

EFFICIENCY & LIMITATIONS

Learning Objectives

- You can illustrate with some examples that there are problems that can be formulated algorithmically but which cannot be solved with the computer.
- You know what is meant by the polynomial and non-polynomial time complexity of a program.
- You can explain the halting problem based on the example of the 3n+1 algorithm.
- You know what is meant by combinatorial explosion.
- You can search a graph using backtracking.
- You know some classic encryption methods and can implement them in a program.
- You know the concept of the finite state machine and know how to implement a simple finite-state machine..

10.1 COMPLEXITY WITH SORTING

INTRODUCTION

Computers are far more than pure calculating machines for simply crunching numbers. Rather, it is known that a considerable portion of the processing time of all computers active worldwide is used for sorting and searching data. It is therefore important that a program not only provides the correct data, but is also optimized. This applies to:

- its length
- its structure and clarity
- its execution time
- its memory usage

Basically, one should already think of optimizing in the beginning of the development process, since it is usually difficult to optimize a sloppily written program afterwards.

In this chapter you will examine the optimization of the execution time when sorting data. You will also get to know the limits of computer science and computer use because a problem, for which there is probably an algorithmic solution process but even the fastest computer would need hundreds of years to solve, is considered to be an **unsolvable problem.**

PROGRAMMING CONCEPTS: *Complexity, execution time, order of algorithms, sorting methods, overloading operators*

SORTING LIKE CHILDREN: CHILDREN SORT

The sorting and ordering of a set of objects with the comparative operations *larger, smaller* and *equal* [**more...**] is and remains a standard task in computer science. Although you will find library routines in all common high-level programming languages that you can sort with, you should include the concepts of sorting to your standard knowledge because there are always situations where you need to implement or optimize the sorting yourself.

A collection of unsorted objects is referred to as a set. The objects are saved in a one-dimensional data structure in the computer where a list is especially well suited [**more...**].

In the program, consider the dwarfs to be actors of the game library *JGameGrid*. You can easily display their sprite images in a grid [**more...**]. The height of the sprite image (in pixels) also serves as a measure of the body size.

Algorithms are often borrowed directly from processes that we also apply in your everyday life. If you ask children how they organize a set of objects by size, they often describe the process as follows: "*You take the smallest (or biggest) object and set it down in order*". This solution process seems very plausible, but there is a problem for the computer because it can not determine the

smallest or biggest object like humans can with just a simple glance. It must first look for the smallest object in the unordered list by running through all the objects in order and comparing them. To implement the sorting process, in this case called **children sort**, you need a function *getSmallest(row)* which returns the smallest dwarf from the reviewed list. You can start as follows:

Save the first list element in the variable *smallest* and run through all subsequent elements in a for-loop. If the element you are looking at is smaller than *smallest*, replace *smallest* with it.

You use two lists for children sort, one list *startList* with the given objects and another list *targetList* which is initially empty. You search for the smallest element in *startList*, remove it from there, and then add it to the back of *targetList* until *startList* is empty.

```
from gamegrid import *
import random

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(startList)):
       addActor(startList[i], Location(i, 0))
   for i in range(len(targetList)):
       addActor(targetList[i], Location(i, 1))

def getSmallest(li):
    global count
    smallest = li[0]
    for dwarf in li:
        count += 1
        if bodyHeight(dwarf) < bodyHeight(smallest):
            smallest = dwarf
    return smallest

n = 7

makeGameGrid(n, 2, 170, Color.red, False)
setBgColor(Color.white)
show()

startList = []
targetList = []

for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    startList.append(dwarf)
random.shuffle(startList)
updateGrid()
setTitle("Children Sort. Press <SPACE> to sort...")
count = 0
while not isDisposed() and len(startList) > 0:
    c = getKeyCodeWait()
    if c == 32:
        smallest = getSmallest(startList)
        targetList.append(smallest)
        startList.remove(smallest)
        count += 1
        setTitle("Count: " + str(count) + " <SPACE> for next step...")
        updateGrid()
setTitle("Count: " + str(count) + " All done")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

With children sort, besides the given unordered list of length *n*, you need a second list that eventually also has the length *n*. If *n* is very large this could lead to a memory problem. [**more...**].

You can easily figure out how many elementary steps are required to solve the problem: Regardless of how the objects are arranged in the given list, you first need to run through all *n* elements of the list, then through n-1, etc. In addition to this, you have to perform the operation of moving the found element from the start list to the target list each time. The number of operations c is therefore the sum of all natural numbers from 2 to n + 1 as you can see with your variable *count*. Using the formula for sums of natural numbers we get:

$$c = \frac{(n+1)*(n+2)}{2} - 1 = \frac{n^2}{2} - \frac{3n}{2}$$

For example, when n = 1000 we already get

$$c = \frac{1000*1000}{2} + \frac{3*1000}{2} = 500000 + 1500 \approx 500000$$

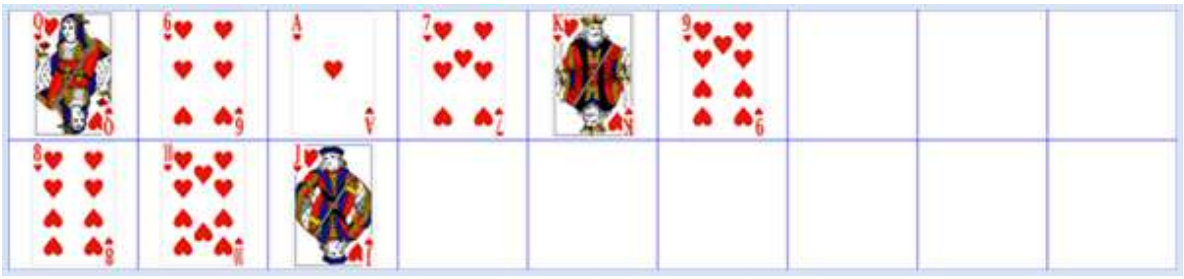
steps! As you can see, the quadratic term prevails for large *n* values and this is why we say
The complexity of the algorithm is of the n-square order
which we write as:

Complexity = O(n^2)

SORTING THE CARD GAME: INSERTION SORT

When you hold the cards in a fan shape while playing cards, you often intuitively use another sorting method: You add each newly obtained card to your 'fan' in a particular place where it fits best according to its value. In your program that inserts the disordered cards from the start list (the deck) into the target list (your hand), you proceed exactly in this way:

You take card by card from left to right from the start list and run though the already ordered target list from the left to right as well. As soon as the card you picked up is considered to be higher in value than the last one considered in your hand, you add it to the target list right there.



```
from gamegrid import *
import random

def cardValue(card):
    return card.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(startList)):
       addActor(startList[i], Location(i, 0))
   for i in range(len(targetList)):
```

```
        addActor(targetList[i], Location(i, 1))

n = 9

makeGameGrid(n, 2, 130, Color.blue, False)
setBgColor(Color.white)
show()

startList = []
targetList = []

for i in range(0 , 9):
    card = Actor("sprites/" + "hearts" + str(i) + ".png")
    startList.append(card)

random.shuffle(startList)
updateGrid()
setTitle("Insertion Sort. Press <SPACE> to sort...")
count = 0

while not isDisposed() and len(startList) > 0:
    getBg().clear()
    c = getKeyCodeWait()
    if c == 32:
        pick = startList[0] # take first
        startList.remove(pick)
        i = 0
        while i < len(targetList) and cardValue(pick) > cardValue(targetList[i]):
            i += 1
            count += 1
        targetList.insert(i, pick)
        count += 1
        setTitle("Count: " + str(count) + " <SPACE> for next step...")
        updateGrid()
setTitle("Count: " + str(count) + " All done")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

This sorting method is called **insertion sort**. The number of steps necessary depends on the order of the cards in the deck. The most steps are needed in a situation where the deck of cards is coincidentally ordered in reverse. One can reflect about it, or find out with a computer simulation, that the number of steps on average (for large *n*) increase with $n^2 / 4$, and thus the average complexity is also $O(n^2)$, as with children sort.

SORTING WITH BUBBLES: BUBBLE SORT

A known way to sort objects in a list is to repeatedly run through the list from left to right and to always swap two adjacent elements if they are in the wrong order.

With this method, first the largest element moves successively from left to right until it has arrived in the final cell. In the next pass you start again on the left, but only go up to the second to last element since the largest is already in place. No second list is necessary in this process [**more...**].

```
from gamegrid import *
import random

def bubbleSize(bubble):
    return bubble.getImage().getHeight()

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))

def exchange(i, j):
    temp = li[i]
    li[i] = li[j]
    li[j] = temp

n = 7
li = []

makeGameGrid(n, 1, 150, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    bubble = Actor("sprites/bubble" + str(i) + ".png")
    li.append(bubble)
random.shuffle(li)
updateGrid()
setTitle("Bubble Sort. Press <SPACE> for next step...")
k = n - 1
i = 0
count = 0
while not isDisposed() and k > 0:
    getBg().fillCell(Location(i, 0), makeColor("beige"))
    getBg().fillCell(Location(i + 1, 0), makeColor("beige"))
    refresh()
    c = getKeyCodeWait()
    if c == 32:
        count += 1
        bubble1 = li[i]
        bubble2 = li[i + 1]
        refresh()
        if bubbleSize(bubble1) > bubbleSize(bubble2):
            exchange(i, i + 1)
            setTitle("Last Action: Exchange. Count: " + str(count))
        else:
            setTitle("Last Action: No Exchange. Count: " + str(count))
        getBg().clear()
        updateGrid()
        if i == k - 1:
            k = k - 1
            i = 0
        else:
            i += 1
getBg().clear()
refresh()
setTitle("All done. Count: " + str(count))
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The largest elements move over to the right, just like bubbles move up in water. Because of this, the name of this sorting algorithm is **bubble sort**. You can think about it, or check the built-in step counter, its complexity is independent of the arrangement of the elements in the

specified list, but again of the order $O(n^2)$.

To make the demonstration a bit more exciting, both cells whose bubbles were compared last are colored using the background method *fillCell()*. The background color can be cleared with *getBg().clear().* You have to call *refresh()* so that the image is re-rendered on the screen.

SORTING WITH LIBRARY ROUTINES: TIMSORT

Since sorting is one of the most important tasks, all high-level programming languages provide built-in library functions for sorting. In *Python*, the function *sorted(list, cmp)* even belongs to the standard built-in functions, which means that it can be used without having to import anything. You can thus save yourself from having to write a sorting algorithm, but in order to do that you have to learn how the library function is used. Clearly it need the list to be sorted as a parameter. With a second parameter, you have to tell it according to which classification criterion it should sort the objects.

You define the sorting criterion in a function which here you call *compare()*. This function has to obtain two objects as parameters and return one of the three values 1, 0, and -1, depending on whether the first object is greater, equal to, or less than the second object. You pass the library function *sorted()* your freely chosen function name as a second parameter or using the named parameter *cmp*.

```
from gamegrid import *
import random

def bodyHeight(dwarf):
    return dwarf.getImage().getHeight()

def compare(dwarf1, dwarf2):
    if bodyHeight(dwarf1) < bodyHeight(dwarf2):
        return -1
    elif bodyHeight(dwarf1) > bodyHeight(dwarf2):
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(li)):
       addActor(li[i], Location(i, 0))

n = 7
li = []

makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Actor("sprites/dwarf" + str(i) + ".png")
    li.append(dwarf)
random.shuffle(li)
updateGrid()
setTitle("Timsort. Press any key to get result...")
getKeyCodeWait()
li = sorted(li, cmp = compare)
updateGrid()
setTitle("All done.")
```

MEMO

If you want to use library functions to sort, you have to specify with a comparison function how two elements are compared to find out whether they are *greater, equal* or *lesser* [**more...**].

The algorithm used in *Python* was invented by Tim Peters in 2002 and is thus called *Timsort*. It has (on average) the order O(nlog(n)). So, for example, when n = 10^6 only about 10^7 operations are necessary instead of about 10^{12} as it would be with a sorting algorithm with the order O(n^2).

EXERCISES

1. Sort the 7 dwarfs with a bubble sort.

2. Add the sprite image *snowwhite.png* of Snow White that has the same size as the largest dwarf into the bubble sort from exercise 1. Show that the order of Snow White and the largest dwarf always correspond to their order in the start list. (Such a sorting algorithm is called **stable**.)

3. You can very easily generate long unsorted lists of numbers using *row = range(n)* and subsequently *random.shuffle(row)*. Measure the execution time of the internal sorting algorithm (*Timsort*) for different values of *n* and show that the complexity is much better than O(n^2). Instructions: In order to measure a time difference, import the module *time* and calculate the difference between two calls of *time.clock()*.

ADDITIONAL MATERIAL

OVERLOADING THE COMPARISON OPERATIONS

Comparing two objects is an important operation. You can use the 5 comparison operations <, <=, ==, >, = > for numbers. In *Python* it is possible to apply these operators to some other data types, for example for dwarfs. Through this, your code gains elegance and clarity.

You can do the following:

In the class definition of your data type, define the methods *_lt_(), __le__(), __eq__(), __ge__(), __gt__()* that return the Boolean values of the comparison operations *less, less-and-equal, equal, greater-and-equal, greater*. In addition, you can also define the method *__str()__*, which is used when the function *str()* is called.

In the class *Dwarf* (derived from *Actor*), you also save the name of the dwarf as an instance variable that you can write out as a *TextActor* upon *updateGrid()*.

```
from gamegrid import *
import random

class Dwarf(Actor):
    def __init__(self, name, size):
        Actor.__init__(self, "sprites/dwarf" + str(size) + ".png")
        self.name = name
        self.size = size
    def __eq__(self, a):  # ==
        return self.size == a.size
    def __ne__(self, a): # !=
        return self.size != a.size
```

```
    def __gt__(self, a): # >
        return self.size > a.size
    def __lt__(self, a): # <
        return self.size < a.size
    def __ge__(self, a): # >=
        return self.size >= a.size
    def __le__(self, a): # <=
        return self.size <= a.size
    def __str__(self):  # str() function
        return self.name

def compare(dwarf1, dwarf2):
    if dwarf1 < dwarf2:
        return -1
    elif dwarf1 > dwarf2:
        return 1
    else:
        return 0

def updateGrid():
   removeAllActors()
   for i in range(len(row)):
       addActor(row[i], Location(i, 0))
       addActor(TextActor(str(row[i])), Location(i, 0))

n = 7
row = []
names = ["Monday", "Tuesday", "Wednesday", "Thursday",
         "Friday", "Saturday", "Sunday"]

makeGameGrid(n, 1, 170, Color.red, False)
setBgColor(Color.white)
show()
for i in range(0 , n):
    dwarf = Dwarf(names[i], i)
    row.append(dwarf)
random.shuffle(row)
updateGrid()
setTitle("Press any key to get result...")
getKeyCodeWait()
row = sorted(row, cmp = compare)
updateGrid()
setTitle("All done.")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The use of comparison operators for arbitrary data types is not mandatory, but elegant. One says that the operators are **overloaded**.

10.2 UNSOLVABLE PROBLEMS

INTRODUCTION

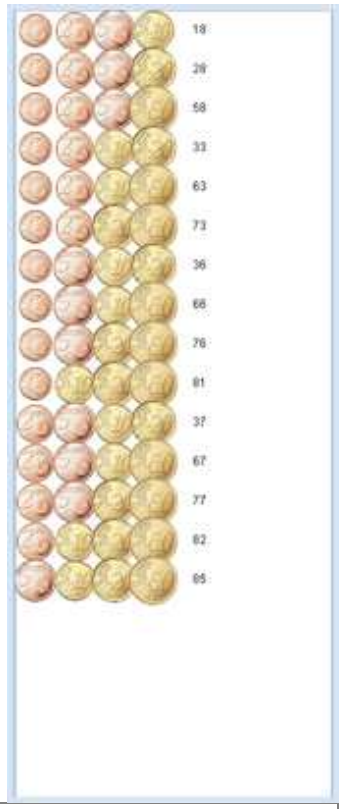
Increasingly many problems can be solved with clever computer programs. In this chapter, however, you will be confronted with questions that can be simply formulated but that may never be algorithmically solvable, despite the rapid evolution of computers and enormous scientific effort.

PROGRAMMING CONCEPTS: *Unsolvable problems, subset sum problem, enumeration methods, combinatorial explosion, polynomial order, undecidable problem*

UNSOLVABLE PROBLEMS

There are still some problems that are unsolved, even though they are easy to formulate and especially important in practice. One of these, known as the **subset sum problem**, can be described as follows [**more...**]:

You have a number of coins in your purse and you must pay a certain amount with it (without getting change back). Is it possible with the existing coins, and if so, what coins should you pay with?



In your first program you will first learn to handle the coins. You save the names of the Euro coins with the values 1, 2, 5, 10, 20, 50 cents in the list *coins*. The function *value()* returns the value of a coin. The purse is modeled as a list (or a tuple) named *moneybag* and contains the names of the coins contained in the purse. The function *getSum(moneybag)* then returns the total value of all coins in the purse.

The purse should initially contain exactly one of each coin. You create all possible variations of coin combinations with 1, 2, 3, 4, 5 or 6 coins and display them in a *JGameGrid* window. To do this, you make an actor out of each coin of *moneybag* in *showMoneybag(moneybag, y)* and display them in the game window in the line *y*.

```
from gamegrid import *
import itertools

coins = ["one", "two", "five", "ten", "twenty", "fifty"]

def value(coin):
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
        return 5
    if coin == "ten":
```

```
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    sum = 0
    for coin in moneybag:
        sum += value(coin)
    return sum

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(8, 20, 40, False)
setBgColor(Color.white)
show()

n = 6
k = 1
while k <= n:
    combinations = list(itertools.combinations(coins, k))
    print type(combinations)
    setTitle("(n, k) = (" + str(n) + ", " + str(k) + ") nb = "
    + str(len(combinations)))
    y = 0
    for moneybag in combinations:
        showMoneybag(moneybag, y)
        y += 1
    getKeyCodeWait()
    removeAllActors()
    k += 1
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The combinations of *k* elements that you can build from a list of *n*, are easily retrievable with the function *combinations()* from the module *itertools*. You have to convert the return value to a list, from which the found combinations can then be retrieved (as tuples).

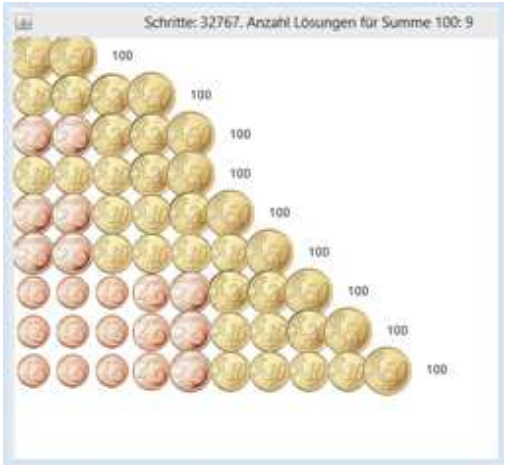
As you can see, the obtained combinations are ordered in a way similar to how you would reasonably order them by hand. You can calculate the number of combinations of *n* elements to the order k as follows, as you know:

$$c = \binom{n}{k} = \frac{n!}{k!*(n-k)!}$$

where n! is the factorial, meaning the product of all numbers from 1 to n. In our case n = 6 can result in 6, 15, 20, 15, 6, 1, so a total of 63 combinations.

You can now solve the subset sum problem of the purse as follows: Determine all combinations of the existing coins and examine them individually to see if their sum adds up to the desired value.

This **enumeration method** is probably not the best to use, but it is definitely correct and it provides all possible solutions. For a purse that contains 3 one-cent, 1 two-cents, 2 five-cents, 4 ten-cents, 2 twenty-cents, and 3 fifty-cents coins (15 coins in total) it would already become difficult to find the solution by hand. You only write all different sets of coins that amount to 1 Euro.




```
from gamegrid import *
import itertools

coins = ["one", "one", "one", "two", "five", "five",
         "ten", "ten", "ten", "ten", "twenty", "twenty",
         "fifty", "fifty", "fifty"]

def value(coin):
    if coin == "one":
        return 1
    if coin == "two":
        return 2
    if coin == "five":
        return 5
    if coin == "ten":
        return 10
    if coin == "twenty":
        return 20
    if coin == "fifty":
        return 50
    return 0

def getSum(moneybag):
    sum = 0
    for coin in moneybag:
        sum += value(coin)
    return sum

def showMoneybag(moneybag, y):
    x = 0
    for coin in moneybag:
        loc = Location(x, y)
        removeActor(getOneActorAt(loc))
        coinActor = Actor("sprites/" + coin + "cent.png")
        addActor(coinActor, loc)
        x += 1
    addActor(TextActor(str(getSum(moneybag))), Location(x, y))

makeGameGrid(15, 20, 40, False)
setBgColor(Color.white)
show()

target = 100

k = 1
result = []
count = 0
while k <= len(coins):
    combinations = tuple(itertools.combinations(coins, k))
```

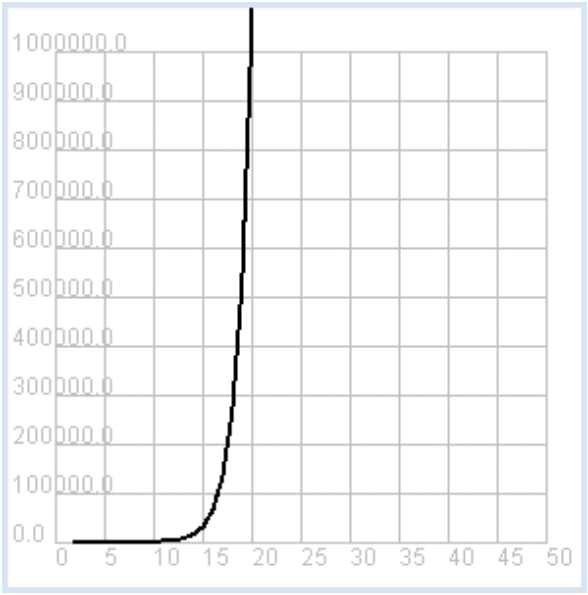
```
    nb = len(combinations)
    for moneybag in combinations:
        count += 1
        sum = getSum(moneybag)
        if sum == target:
            if not moneybag in result:
               result.append(moneybag)
    k += 1

y = 0
for moneybag in result:
    showMoneybag(moneybag, y)
    y += 1
setTitle("Step: " + str(count) + ". number of solutions for the sum  "
          + str(target) + ": " + str(len(result)))
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

With only 15 coins there are already 32,767 steps necessary in order to solve the subset sum problem using the enumeration method.

Your enjoyment of solving problems with the computer is unfortunately spoiled once you try with a slightly larger purse of money, let's say 50 or 100 coins. If you count the necessary steps it takes for a purse with *n* coins and display them in a graph, there is a downright **combinatorial explosion** at n = 20 and you reach the limit of what is possible [**more...**].




```
from gpanel import *
from math import factorial

z = 100

def nbCombi(n, k):
    return factorial(n) / factorial(k) / factorial(n - k)

makeGPanel(-5, 55, -1e5, 1.1e6)
drawGrid(0, 50, 0, 1e6, "gray")
setColor("black")
lineWidth(2)
for n in range(2, z + 1):
    sum = 0
    for k in range(1, n):
        sum += nbCombi(n, k)
```

```
    print "n =", n, ", nb =", sum
    if n == 2:
        move(n, sum)
    else:
        draw(n, sum)
print "Runtime with 10^9 operations per second:", sum / 3.142e16, "years"
print "or:", int(sum / 2e20), "times the age of the universe"
```

MEMO

Using the enumeration method, the subset sum problem is already **unsolvable** at relatively small amounts of elements, even though the method of solving is known. Hence the question arises whether there could be **much better algorithms** to solve the problem, the number of steps or complexity of which is a power of *n* (thus **polynomial**), just as the ones in the previous chapter for sorting. Unfortunately, until today no one has succeeded in finding such an algorithm for the subset sum problem and one assumes that there is none. However, there is also no theoretical proof for this assumption.

At least we know today from the theoretical computer science that there are many **similarly difficult problems** and that one could expect to solve all these problems in one shot with polynomial complexity, if one finds such a solution for one of them.

UNDECIDABLE PROBLEMS

The limits of the human mind and computer technology are also visible in a context different from complexity. The mathematician and number theorist Lothar Collatz examined certain sequences of natural numbers and in 1939 he formulated the following question:

Begin from any initial number *n* and build the following numbers according to these rules:

- If n is even, divide n by 2 (a natural number again)
- If n is odd, build the following number 3n +1 (an even number)

Question: Does this sequence always reach 1 for any possible starting number *n*?

Collatz and many other number theorists and computer scientists have searched for a solution to this, because even the largest and fastest computers continuously yield sequences that arrive at the result 1. (The series does not converge because it endlessly repeats through the sequence 4, 2, 1).

Therefore, it seems **likely** that the following theorem applies:

The 3n+1 series reaches the number 1 for all natural starting numbers n after a finite number of steps.

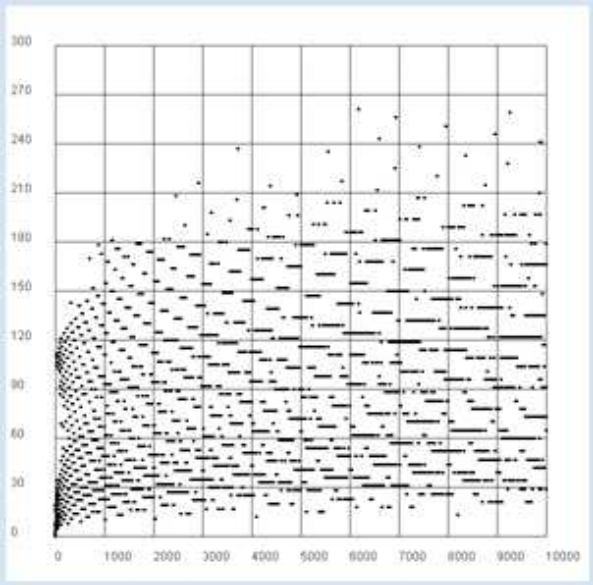
You can run through the 3n+1 series with a computer program for an arbitrarily settable starting number.

```
from gpanel import *

def collatz(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        print n,
    print "Result 1"
while True:
    n = inputInt("Enter a start number:")
    collatz(n)
```

MEMO

In *Python* you can even run through the 3n+1 series with large starting numbers and you will find that you always land at 1. Of course you have not proven the assumption this way.

It is interesting and aesthetically appealing to plot the length of the 3n+1 series in relation to the starting number. It fluctuates quite considerably. To do this, remove the writing out of the terms in the function *collatz()* and only return the number of steps.




```
from gpanel import *

def collatz(n):
    nb = 0
    while n != 1:
        nb += 1
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
    return nb

z = 10000 # max n
yval = [0] * (z + 1)
for n in range(1, z + 1):
    yval[n] = collatz(n)
ymax = (max(yval) // 100  + 1) * 100

makeGPanel(-0.1 * z, 1.1 * z, -0.1 * ymax, 1.1 * ymax)
title("Collatz Assumption")
drawGrid(0, z, 0, ymax, "gray")

for x in range(1, z + 1):
    move(x, yval[x])
    fillCircle(z / 200)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Collatz's assumption is a stubborn problem. If the assumption really is true, then it cannot be proved by conducting computer tests with increasingly larger starting numbers. It is even possible that the assumption is true, but a proof will never be found. In 1931 the mathematician Kurt Gödel showed with the **incompleteness theorem** that there can be correct propositions in a theory, whose correctness can, however, not be proven.

Collatz's assumption can also be formulated as a **decision problem**:

Does an algorithm that calculates the terms of the 3n+1 progression and stops at 1, really stop for any possible initial values?

You could try to solve this question with a computer. Unfortunately, this could be hopeless too, since the great mathematician and computer scientist Alain Turing has already proved with the **halting problem** that there will never be a general algorithm that allows you to decide if any given program with arbitrary input really stops.

Collatz's assumption of 3n+1 could thus indeed be true, but an **undecidable problem**.

10.3 BACKTRACKING

INTRODUCTION

In the development of computer games, it only gets really interesting when the computer itself becomes an intelligent game partner. Such a program must not only comply with the rules of the game, but it must also pursue a winning strategy. In order to implement a game strategy, the game should be understood as a sequence of game situations that can be clearly identified with a suitable variable *s*. These are known as **game states** and is therefore *s* is called a **state variable**. The goal of the strategy is to move from an initial state to a winning state where the game is usually over.

The game states can be neatly shown as **nodes** in a **game graph**. For each turn, there is a transition from one node to one of its successors. The rules of the game specify what the possible successor nodes or **neighbors** of a particular game state are. You can draw the transition as a directional connecting line, also called an **edge** [**more...**].

In this section you will learn important techniques that have a general validity and help you to write computer games that can win even against very intelligent human players. However, in addition to these general techniques there is still plenty of room for your own ideas, to write more efficient, simpler, and better adapted game strategies that possibly have a better chance of winning or consumes less computing time.

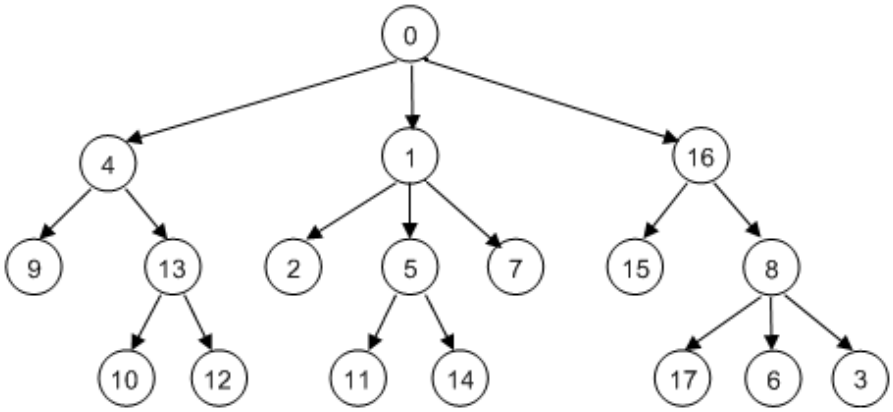
It is a fact that many political, economical, and social systems can be understood as a game, and so you can apply your acquired knowledge on a wide array of practically relevant fields.

PROGRAMMING CONCEPTS: *Game state, game graph, depth-first search, backtracking*

SEARCHING FOR A SOLUTION FOR A SOLO GAME

As previously mentioned, you represent the game states as nodes in a graph that is run through step by step. You have to first uniquely determine the game states by certain criteria, such as the arrangement of the tokens on a game board. The rules of the game specify which are the possible successor nodes or neighbors for a specific game state. These are connected with the node by lines (edges). Since they are successor node, the edges have a direction from the node to its neighbors. Sometimes a node is also called a "mother" and its neighbors "daughters", and there is also the possibility that there is an edge from a daughter back to her mother.

Here you begin from a simple game graph for a game that is played either by a single person or by the computer alone. The single person game, or solo game, should be constructed so that there are no paths that lead back again. This ensures that you do not fall into a cycle that runs endlessly when moving through the graph. Such a special graph is called a **tree** [**more...**]. You can identify the nodes in any order with numbers between 0 and 17. The graph has the following structure:



The tree should be saved as a whole in a suitable data structure. A list is well suited for this, in which the numbers of neighboring nodes are included as sublists where at the index 0 we find the list of neighbors of node 0, at index 1 is the list of neighbors of node 1, etc. If a node does not have a neighbor, its neighbor list is empty [**more...**].

As you can easily see, the following list represents the given tree

```
neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13],[11, 14], [], [], [17, 6, 3],
[], [], [], [], [10, 12], [], [], [15, 8], []]
```

Identifying a node by a number is a trick that allows you to determine the neighbors of a node with a list index. The algorithm for finding the path from a certain node to one in the deeper tree structure is defined recursively in the function *search(node)*. Formulated in **pseudo code** it reads:

```
search(node):
   if node == targetnode:
        print "Target achieved"
        return
    determine list of neighbors
    run through this list and execute:
        search(neighbors)
```

Additionally, the "visited" nodes are entered in the back of the list *visited*. If the goal is not reached earlier, the node is removed again from *visited* after all nodes were traversed, in order to go back to the original state [**more...**]. The start and target node number can be entered at the beginning of the program.

```
neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

def search(node):
    visited.append(node) # put (push) to stack

    # Check for solution
    if node == targetNode:
        print "Target ", targetNode, "achieved. Path:", visited
        targetFound = True
        return

    for neighbour in neighbours[node]:
        search(neighbour) # recursive call
    visited.pop() # redraw (pop) from stack

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Start node (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
```

```
   targetNode = inputInt("Target node (0..17):")
visited = []
search(startNode)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

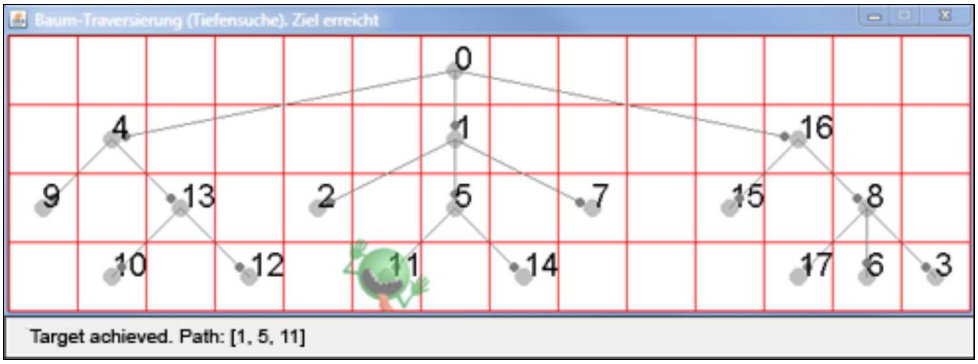
MEMO

The correct path [0, 1, 5, 14] is written out for the start node 0 and the target node 14. If you add the node 0 as a neighbor at node 13, it results in a disastrous situation and the program is aborted with a runtime error that says that the maximum recursion depth is reached.

THE TRAVERSING OF AN ALIEN

It is neat to be able to visibly follow the algorithm by representing the game tree graphically and gradually traversing it (by pressing a key). For this you best use a *GameGrid* window in which nodes are made visible as circles in certain cell coordinates (locations).

In the current cell you see a semi-transparent alien waving at you.



You draw the tree with the graphical methods of *GGBackground*. You can attach a small circle mark to the edges, instead of an arrow tip, using *getMarkerPoint()* to show the direction of the edge. Make sure that you refresh the screen using *refresh()*. You can view important information in the status bar.

```
from gamegrid import *

neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

locations = [Location(6, 0), Location(6, 1), Location(4, 2), Location(13, 3),
             Location(1, 1), Location(6, 2), Location(12, 3), Location(8, 2),
             Location(12, 2), Location(0, 2), Location(1, 3), Location(5, 3),
             Location(3, 3), Location(2, 2), Location(7, 3), Location(10, 2),
             Location(11, 1), Location(11, 3)]

def drawGraph():
    getBg().clear()
    for i in range(len(locations)):
        getBg().setPaintColor(Color.lightGray)
        getBg().fillCircle(toPoint(locations[i]), 6)
        getBg().setPaintColor(Color.black)
        getBg().drawText(str(i), toPoint(locations[i]))
        for k in neighbours[i]:
            drawConnection(i, k)
    refresh()

def drawConnection(i, k):
    getBg().setPaintColor(Color.gray)
    startPoint = toPoint(locations[i])
```

```
    endPoint = toPoint(locations[k])
    getBg().drawLine(startPoint, endPoint)
    getBg().fillCircle(getMarkerPoint(endPoint, startPoint, 10), 3)

def search(node):
    global targetFound
    if targetFound:
        return
    visited.append(node) # put (push) to stack
    alien.setLocation(locations[node])
    refresh()
    if node == targetNode:
        setStatusText("Target " + str(targetNode) + "achieved. Path: "
                      + str(visited))
        targetFound = True
        return
    else:
        setStatusText("Current node " + str(node) + " .  Visited: "
        + str(visited))
    getKeyCodeWait(True) # exit if GameGrid is disposed

    for neighbour in neighbours[node]:
        search(neighbour)  # Recursive call
    visited.pop()

makeGameGrid(14, 4, 50, Color.red, False)
setTitle("Tree-traversal (depth-first search). Press a key...")
addStatusBar(30)
show()
setBgColor(Color.white)
drawGraph()

startNode = -1
while startNode < 0 or startNode > 17:
   startNode = inputInt("Start node  (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
   targetNode = inputInt("Target node  (0..17):")

visited = []
targetFound = False
alien = Actor("sprites/alieng_trans.png")
addActor(alien, locations[startNode])

search(startNode)
setTitle("Tree-traversal (depth-first search). Target achieved")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

As you can see, the alien moves to the daughter nodes "in the depth of the tree" and then jumps back to the last mother node. For this reason, this principle is called **depth-first search with backtracking**.

THE ALIEN ON THE WAY BACK

If you want to make visible the path that the alien moves while moving back, you need to save the sequence of nodes while moving forward in a list *steps*. There is a new list at each recursion depth and you save these in *stepsList*. After moving back, you have to remove the last entry.

```
from gamegrid import *
```

```
neighbours = [[4, 1, 16], [2, 5, 7], [], [], [9, 13], [11, 14], [], [],
              [17, 6, 3], [], [], [], [], [10, 12], [], [], [15, 8], []]

locations = [Location(6, 0), Location(6, 1), Location(4, 2), Location(13, 3),
             Location(1, 1), Location(6, 2), Location(12, 3), Location(8, 2),
             Location(12, 2), Location(0, 2), Location(1, 3), Location(5, 3),
             Location(3, 3), Location(2, 2), Location(7, 3), Location(10, 2),
             Location(11, 1), Location(11, 3)]

def drawGraph():
    getBg().clear()
    for i in range(len(locations)):
        getBg().setPaintColor(Color.lightGray)
        getBg().fillCircle(toPoint(locations[i]), 6)
        getBg().setPaintColor(Color.black)
        getBg().drawText(str(i), toPoint(locations[i]))
        for k in neighbours[i]:
            drawConnection(i, k)
    refresh()

def drawConnection(i, k):
    getBg().setPaintColor(Color.gray)
    startPoint = toPoint(locations[i])
    endPoint = toPoint(locations[k])
    getBg().drawLine(startPoint, endPoint)
    getBg().fillCircle(getMarkerPoint(endPoint, startPoint, 10), 3)



def search(node):
    global targetFound
    if targetFound:
        return
    visited.append(node) # put (push) to stack
    alien.setLocation(locations[node])
    refresh()
    if node == targetNode:
        setStatusText("Target " + str(targetNode) + "achieved. Path: "
                      + str(visited))
        targetFound = True
        return
    else:
        setStatusText("Current nodes " + str(node) + " .  Visited: "
                      + str(visited))
    getKeyCodeWait(True) # exit if GameGrid is disposed

    for neighbour in neighbours[node]:
        steps = [node]
        stepsList.append(steps)
        steps.append(neighbour)
        search(neighbour)  # Recursive call
        steps.reverse()
        if not targetFound:
            for loc in steps[1:]:
                setStatusText("Go back")
                alien.setLocation(locations[loc])
                refresh()
                getKeyCodeWait()
        stepsList.pop()
    visited.pop()

makeGameGrid(14, 4, 50, Color.red, False)
setTitle("Tree-traversal (depth-first search). Press a key...")
addStatusBar(30)
show()
setBgColor(Color.white)
drawGraph()

startNode = -1
```

```
while startNode < 0 or startNode > 17:
   startNode = inputInt("Start node  (0..17):")
targetNode = -1
while targetNode < 0 or targetNode > 17:
   targetNode = inputInt("Target node (0..17):")

visited = []
stepsList = []
targetFound = False
alien = Actor("sprites/alieng_trans.png")
addActor(alien, locations[startNode])

search(startNode)
setTitle("Tree-traversal (depth-first search). Target  achieved")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The alien now really moves back up in the tree, which makes it particularly clear why the algorithm is called **backtracking**. Recursive backtracking plays an important role in many algorithms and is sometimes also referred to as the "Swiss army knife of computer scientists".

STRATEGY IN A MAZE

Sometimes it is difficult, or even just frightening, to find your way out of a maze. However, now you can write a program with your knowledge about backtracking that can find its way out of a maze with certainty. It is quite obvious that you can model a maze that has no circles as a tree. Finding the exit in such a maze therefore corresponds a tree traversal.

Here you use only a small random maze with 11x11 cells. The alien moves one step when you press the button, but if you hit the Enter key it searches for the exit completely autonomously.

You generate the maze with the class *Maze*. You pass it the desired number of rows and columns as odd numbers. Each time, a different random maze is created with an entry at the top left side and an exit at the bottom right. You can test whether there is a wall cell at the location *loc* using *isWall(loc)*.

It is often not suitable to determine the full game graph before the game. In many cases this is simply impossible, since there are so many game situations that you are not able to determine them in a reasonable time and there would also not be enough space for the data. Due to this, it is usually better to determine the neighboring nodes of the current node only during backtracking.

You detect the neighboring nodes in your program by selecting the 4 adjacent cells that are not a wall or outside of the grid.

```
from gamegrid import *

def createMaze():
    global maze
    maze = GGMaze(11, 11)
    for x in range(11):
        for y in range(11):
            loc = Location(x, y)
```

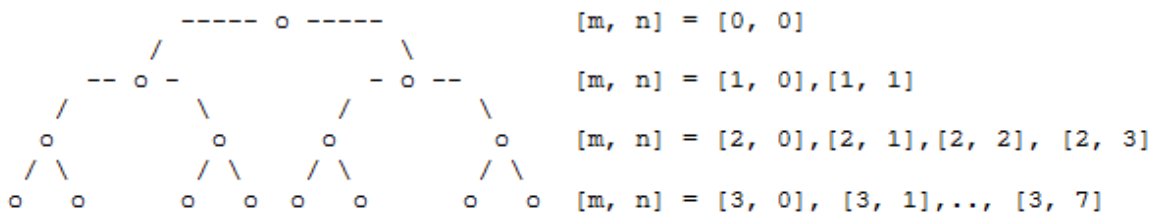
```
            if maze.isWall(loc):
                getBg().fillCell(loc, Color(0, 50, 0))
            else:
                getBg().fillCell(loc, Color(255, 228, 196))
    refresh()

def getNeighbours(node):
    neighbours = []
    for loc in node.getNeighbourLocations(0.5):
        if isInGrid(loc) and not maze.isWall(loc):
           neighbours.append(loc)
    return neighbours

def search(node):
    global targetFound, manual
    if targetFound:
        return
    visited.append(node)  # push
    alien.setLocation(node)
    refresh()
    delay(500)
    if manual:
        if getKeyCodeWait(True) == 10:  #Enter
            setTitle("Finding target...")
            manual = False

    # Check for termination
    if node == exitLocation:
        targetFound = True

    for neighbour in getNeighbours(node):
        if neighbour not in visited:
            backSteps = [node]
            backStepsList.append(backSteps)
            backSteps.append(neighbour)

            search(neighbour) # recursive call

            backSteps.reverse()
            if not targetFound:
                for loc in backSteps[1:]:
                    setTitle("Must go back...")
                    alien.setLocation(loc)
                    refresh()
                    delay(500)
                if manual:
                    setTitle("Went back. Press key...")
                else:
                    setTitle("Went back. Find target...")
            backStepsList.pop()
    visited.pop() # pop

manual = True
targetFound = False
visited = []
backStepsList = []
makeGameGrid(11, 11, 40, False)
setTitle("Press a key. (<Enter> for automatic")
show()
createMaze()
startLocation = maze.getStartLocation()
exitLocation = maze.getExitLocation()
alien = Actor("sprites/alieng.png")
addActor(alien, startLocation)
search(startLocation)
setTitle("Target found")
```

MEMO

It is interesting to compare the solution strategy of a human with that of the computers. A human player can recognize the overall situation with a simple glance and derive a strategy of how to reach the exit as quickly as possible. They act with a characteristically human global overview that your computer program lacks. Your program only detects the current situation locally, but it *"remembers"* the already passed routes very precisely and then systematically searches for new routes leading to the target destination. However, the situation changes in favor of the computer when the human is withheld a global overview, e.g. if they themselves were actually located inside of the maze.

EXERCISES

1. A **binary tree** has two neighboring nodes for every node, namely a left and a right one. Choose a list with two numbers [m, n] as a node identifier, where *m* indicates the depth in the tree and *n* indicates the width.



   The program should write out the path after entering the start and target nodes.

2. It is amazing that you can always find the way out of a maze using the **right-hand rule**, even if it has circles in it. Following it, you always walk with your right hand along the wall and stick to the following rules:

   - If the right is free, then go to the right
   - If you can not go to the right, but going straight is an option, then go straight
   - If you can not go to the right or straight ahead, then turn to the left

   Implement this rule for a *GameGrid* maze with a rotatable beetle Actor *lady = Actor(True, "sprites/ladybug.gif").* Instructions: Place the beetle in the next cell according to the rule using *move()*. If it is a wall cell, undo the step.

   Compare this solution algorithm with the solution you get by backtracking.

ADDITIONAL MATERIAL

THE N-QUEENS PROBLEM

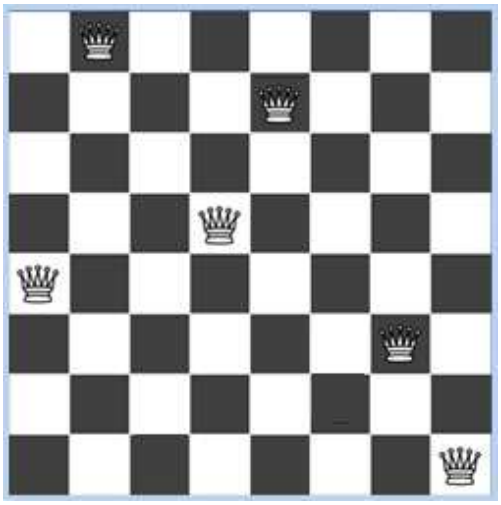
The following is a chess problem that has already been discussed since the mid-19th century. Place *n* queens on a chessboard with *nxn* fields so that they can not beat each other in accordance with the rules of chess. (The rules of chess state that a queen can move both horizontally and vertically, as well as diagonally.) There are two issues with varying degrees of difficulty: On the one hand, you would like to specify a solution, and on the other hand, you would like to find out how many solutions there are in total. Already in 1874 the mathematician Glaisher proved that there are a total of 92 solutions for the classic checkerboard with n = 8.

The queens problem is considered to be a classic example of backtracking. You place the queens one after another onto the board always making sure that an added queen does not come into conflict with the queens already present. If you proceed aimlessly, there is usually a moment where it is no longer possible to place a queen. The applied strategy in backtracking is that it undoes the

last step and tries an alternative. If there is still no solution from the alternative step, you must undo this step as well, etc. With this procedure, a person can easily loose the overview of which positions have already been tested, but a computer on the other hand, does not have this problem.

As with all tasks in backtracking, you can regard the game states as nodes in a game graph. Selecting a suitable data structure is very important. Proceeding in accordance with the "brute force" principle, where you set *n* queens on the board in all possible ways and after sort out those cases which can not be beat, is not suitable because with n = 8 there are around 422 million possible positions.

It is much better if from the beginning you only consider positions where the placed queens are located on different rows and columns. For n = 8 you can specify the game state with a list of 8 numbers, where the first number is the column index of the queen in the first row, the second number is the column index of the queen in the second row, etc. For rows without queens, write -1 as an index. If you take row and column indices from 0 to n-1, you identify the adjacent position with *node = [1, 4, -1, 3, 0, 6, -1, 7].*



You can determine the neighboring nodes of the current nodes (*node*) in the backtracking algorithm using the function *getNeighbours(node)*.

Thereby you switch from the one-dimensional data structure to *Locations*, which uses *x* and *y* coordinates of the fields. You gather the already occupied fields in the list *squares* and you put those which cannot be filled due to the rules of the game in the list *forbidden*. (In this case it is convenient to use the method *getDiagonalLocations()*.) Finally, you create the list *allowed* for fields that can still be filled. You now need to replace the -1 in the neighboring nodes list with the column index on which the new queen is placed. You implement the already known backtracking algorithm in *search()*. Once a solution is found, you stop the search (recursion stop).

```
from gamegrid import *

n = 8 # number of queens

def getNeighbours(node):
    squares = [] # list of occupied squares
    for i in range(n):
        if node[i] != -1:
           squares.append(Location(node[i], i))

    forbidden = [] # list of forbidden squares
    for location in squares:
        a = location.x
        b = location.y
        for x in range(n):
            forbidden.append(Location(x, b))  # same row
        for y in range(n):
            forbidden.append(Location(a, y))  # same column
        for loc in getDiagonalLocations(location, True):   #diagonal up
            forbidden.append(loc)
        for loc in getDiagonalLocations(location, False):  #diagonal down
            forbidden.append(loc)

    allowed = [] # list of all allowed squares = all - forbidden
    for i in range(n):
        for k in range(n):
            loc = Location(i, k)
            if not loc in forbidden:
                allowed.append(loc)
```

```
    neighbourNodes = []
    for loc in allowed:
        neighbourNode = node[:]
        i = loc.y # row
        k = loc.x # col
        neighbourNode[i] = k
        neighbourNodes.append(neighbourNode)
    return neighbourNodes

def search(node):
    global found
    if found or isDisposed():
        return
    visited.append(node)  # node marked as visited

    # Check for solution
    if not -1 in node:
        found = True
        drawNode(node)

    for s in getNeighbours(node):
        search(s)
    visited.pop()

def drawBoard():
    for i in range(n):
        for k in range(n):
            if (i + k) % 2 == 0:
                getBg().fillCell(Location(i, k), Color.white)

def drawNode(node):
    removeAllActors()
    for i in range(n):
        addActorNoRefresh(Actor("sprites/chesswhite_1.png"), Location(node[i], i))
    refresh()

makeGameGrid(n, n, 600 // n, False)
setBgColor(Color.darkGray)
drawBoard()
show()
setTitle("Searching. Please wait..." )

visited = []
found = False
startNode = [-1] * n  # all squares empty
search(startNode)
setTitle("Search complete. ")
```

MEMO

Depending on the performance of your computer, you may have to wait anywhere from a few seconds to a few minutes until a solution is found. If it takes too long, you can set n = 6.

EXERCISES

1. Solve the same task without using the *GameGrid* library. Substitute *Location* with cell lists [i, k]. You can write out the solution as a node list.

2. Generalize the program declared above for n = 6 or your own program from exercise 1 so that all solutions are found. Take into account that a solution node is found several times and avoid duplicates with a list of already found solutions.

10.4 SHORTEST PATH, 3 JUGS PROBLEM

INTRODUCTION

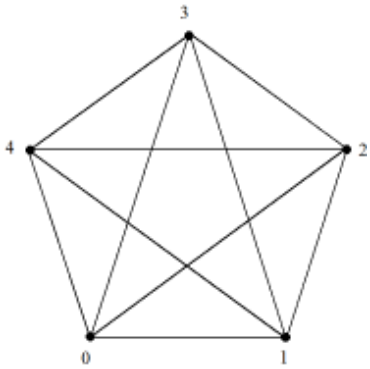
Many important algorithmic solution methods were developed from the practice of daily life, including backtracking. As you have already learned, the computer chooses a game move from all the possibilities of allowed moves and consequently pursues it. If the computer runs into a dead end, it undoes previous moves. This solution strategy is called **trial and error** in the context of daily life, and it is known that it is not always optimal. It would be better to choose the next move as favorably as possible depending on the goal [**more...**].

PROGRAMMING CONCEPTS: *Trial and error, graph with circles, backtracking*

GRAPH WITH CIRCLES

When traversing a graph, it may happen that you arrive back at the same state after making a few steps. Think, for example, of the subway network of a big city where the stations have an interwoven structure. If you want to ride from point A to point B in that city, there are many different options and you could easily end up riding around in circles. In preparation for such a navigation task, you look at a graph with 5 nodes where each node is connected to every other node.

The nodes are identified by a node number 0..4. As you have already seen, it is possible that the simple backtracking algorithm cannot find the path from a given node to another because it gets stuck in a cycle. However, you only need a small supplement to avoid this: Before you make the recursive call to *search()*, check in the list *visited* to see if the concerned neighboring nodes were already visited. If so, you can skip these neighbors. Now all 16 possible routes between nodes are written out in the program.




```
def getNeighbours(node):
    return range(0, node) + range(node + 1, 5)

def search(node):
    global nbSolution
    visited.append(node)  # node marked as visited

    # Check for solution
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Route:", visited
        # don't stop to get all solutions

    for neighbour in getNeighbours(node):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

startNode = 0
targetNode = 4
nbSolution = 0
visited = []
search(startNode)
```

MEMO

By checking if a neighbor has already been visited, you can also make use of backtracking on graphs with cycles. If you forget to check this, your program will "hang", resulting in a bad run-time error due to overflowing of the function call memory.

SHORTEST PATH, NAVIGATION SYSTEMS

Trying to find a path from starting point A to a destination B is omnipresent in daily life. Since there are often many routes leading from A to B (not only to Rome), it is usually also important to determine a certain criterion (route length, traveling time, road quality, landmarks, cost, etc.) in order to find the optimal path [**more...**].

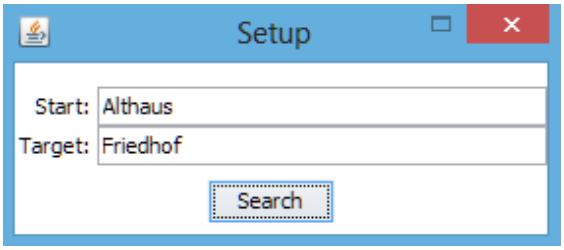
In your program, we will simply focus on the basics. Therefore, you choose a local network with only 6 places that you can regard as subway stations in a fictional city. The nodes of the graph are identified by the names of the stations. (The first letters of these names are A, B, C, D, E, F, so the stations could also be identified with these letters or with node numbers.) The indication of the neighboring stations is an allocation of a station name to a list of of names, which is why a dictionary with the station name as a key and the list of the neighboring stations as a value is perfectly suited for this. Instead of using a function *getNeighbours()*, you directly use a dictionary *neighbours*.

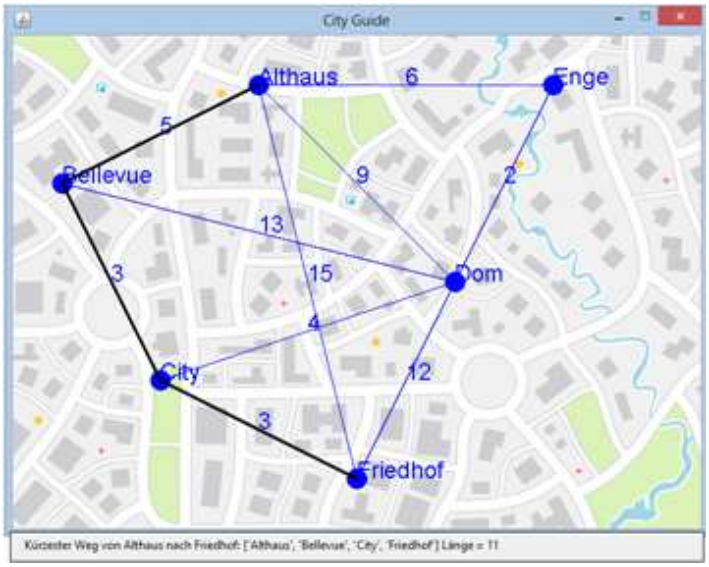
The distance between the stations are also stored analogously in a dictionary *distances* that has the two connected stations as a key and the distance as a value.

In order to enter these stations into a *GameGrid*, you still need a dictionary *locations* with the locations (x, y coordinates) of the stations.

The central part of your program is an exact replication of the backtracking algorithm used above. In addition, you need some auxiliary functions to represent it graphically.

You use an entry dialog for the user input and write the outputs to a status bar. Furthermore, you draw the optimal path in the graph of stations.





```
from gamegrid import *

neighbours = {
 'Althaus':['Bellevue', 'Dom', 'Enge'],
 'Bellevue':['Althaus', 'City', 'Dom'],
 'City':['Bellevue', 'Dom', 'Friedhof'],
 'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
 'Enge':['Althaus', 'Dom'],
 'Friedhof':['Althaus', 'City', 'Dom']}

distances = {
   ('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
   ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
   ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
   ('City', 'Dom'):4, ('City', 'Friedhof'):3,
   ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

locations = {
 'Althaus':Location(2, 0),
 'Bellevue':Location(0, 1),
 'City':Location(1, 3),
 'Dom':Location(4, 2),
 'Enge':Location(5, 0),
 'Friedhof':Location(3, 4)}

def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    sum = 0
    for i in range(len(li) - 1):
        sum += getNeighbourDistance(li[i], li[i + 1])
    return sum

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                   getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    # Check for solution
    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
```

```
            trackToTarget = visited[:]

    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

makeGameGrid(7, 5, 100, None, "sprites/city.png", False)
setTitle("City Guide")
addStatusBar(30)
show()
init()
startStation = ""
while not startStation in locations:
    startStation = inputString("Start station")
targetStation = ""
while not targetStation in locations:
    targetStation = inputString("Target station")
search(startStation)
setStatusText("Shortest way from " + startStation + " to " + targetStation
    + ": " + str(trackToTarget) + " Length = " + str(trackLength))
for i in range(len(trackToTarget) - 1):
    s1 = trackToTarget[i]
    s2 = trackToTarget[i + 1]
    getBg().setPaintColor(Color.black)
    getBg().setLineWidth(3)
    drawConnection(s1, s2)
refresh()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The search for the shortest path in a graph is one of the basic tasks in computer science. The solution shown here achieves its goal with backtracking, but it is very CPU-intensive (meaning it takes a lot of processing power). There are much better algorithms for finding the shortest path where you do not have to systematically search for every route. The most famous is called "Dikjstra's algorithm".

THE THREE JUGS PROBLEM

Brain teasers in which a given quantity has to be divided into certain partial quantities by measuring (pouring, weighting, etc.) have already been found in children's books and magazines for centuries. The well-known three jugs problem is attributed to the French mathematician Bachet de Méziriac in the 17th century and goes as follows:

Two friends have decided to evenly divide up 8 liters of wine that is in an 8 liter jug by pouring it. In addition to the 8 liter jug, you have a 5 liter and a 3 liter jug. The jugs have no markings on them for measuring the contents. How should you proceed and how many pours are necessary, at minimum?

According to the problem description, it is not only a matter of finding the solution, which you might be able to do by thinking a bit, but finding all possible solutions, and therefore determining the shortest of them all. Without a computer, this would be very tiring. Searching for all solutions, such as in this example, is called an **exhaustive search**.

To begin with, you again proceed according to the previously tried solution strategy with backtracking. First invent a suitable data structure for the game states. For this, you use a list with three numbers that describes the current fill level of the three jugs. [1, 4, 3] should thus mean that the 8 liter jug currently contains 1 liter of wine, the 5 liter jug contains 4 liters, and the 3 liter jug contains 3 liters.

You can again model the game states as nodes in a graph and consider the pouring as a transition from one node to its neighboring node. Here, as in many other examples, it does not make sense to construct the entire game tree at the beginning. Instead, you can determine the neighboring nodes of a node *node* in the function *getNeighbours(node)* only when you actually need them during the game. You can begin with the following consideration:

Regardless of how much wine is in the jugs, there are basically 6 options of how you can pour: You take one of the jugs and either pour all of their contents or as much as there is space available in the second jug. You therefore collect the neighboring nodes of these 6 cases in the list *neighbours in getNeighbours()*. The function *transfer(state, source, target)* helps you to figure out the neighboring states of a particular state *state* and given jug numbers *source* and *target* after pouring from *source* to *target.* The jug sizes (maximum capacity) and the already contained quantity will be considered.
Again your recursive function *search()* uses the backtracking algorithm *as you know it.*

```
def transfer(state, source, target):
    # Assumption: source, target 0..2, source != target
    s = state[:] # clone
    if s[source] == 0 or s[target] == capacity[target]:
        return s  # source empty or target full
    free = capacity[target] - s[target]
    if s[source] <= free: # source has enough space in target
        s[target] += s[source]
        s[source] = 0
    else:  # target is filled-up
        s[target] = capacity[target]
        s[source] -= free
    return s

def getNeighbours(node):
# returns list of neighbours
    neighbours = []
    t = transfer(node, 0, 1) # from 0 to 1
    if t not in neighbours:
       neighbours.append(t)
    t = transfer(node, 0, 2) # from 0 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 1, 0) # from 1 to 0
    if  t not in neighbours:
        neighbours.append(t)
```

```
    t = transfer(node, 1, 2) # from 1 to 2
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 0) # from 2 to 0
    if t not in neighbours:
        neighbours.append(t)
    t = transfer(node, 2, 1) # from 2 to 1
    if t not in neighbours:
        neighbours.append(t)
    return neighbours

def search(node):
    global nbSolution
    visited.append(node)

    # Check for solution
    if node == targetNode:
        nbSolution += 1
        print nbSolution, ". Route:", visited, ". Length:", len(visited)

    for s in getNeighbours(node):
        if s not in visited:
            search(s)
    visited.pop()

capacity = [8, 5, 3]
startNode = [8, 0, 0]
targetNode = [4, 4, 0]
nbSolution = 0
visited = []
search(startNode)
print "Done. Find the best  solution!"
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The solutions are written out to the output window. They are not published here so that you are able to approach the problem with an open mind and possibly first try to find the solution with pencil and paper. After they are revealed, you will see that there are 16 solutions, and 16 pours are necessary for the longest.

EXERCISES

1. Simplify the navigation program so that the nodes are identified by the numbers 0, 1, 2, 3, 4, 5 and *neighbours* is a list with sublists.

2. You should scoop exactly 4 liters of water from a lake with a 3 liter and 5 liter jug. Describe how you would proceed and find the shortest amount of pours it would take. Keep in mind that you can pour the water back into the lake again

3. Invent a solvable pouring problem and give it to other people in your community as a brain teaser.

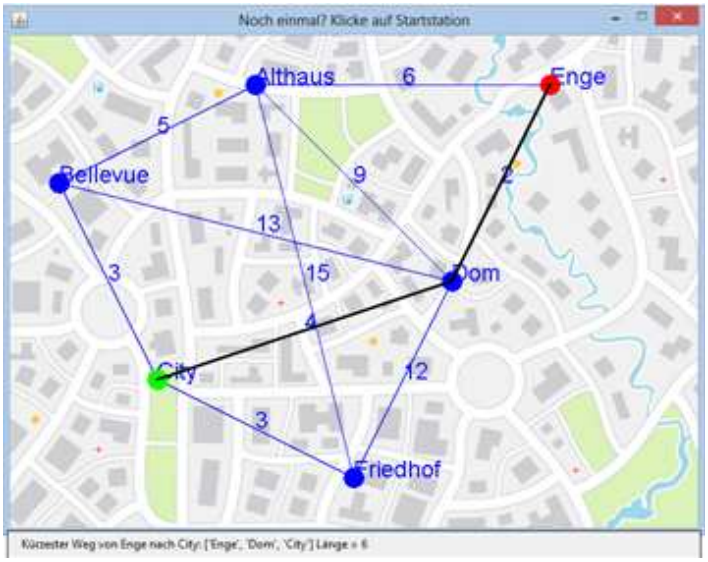
CITY NAVIGATION WITH MOUSE SUPPORT

The graphical user interface plays a central role in any professional program. While designing it, the programmer has to be guided less by programmatic considerations but rather by putting themselves into the shoes of an unbiased user who uses the program with as little effort as possible and natural human logic. Today such an interface usually includes a graphical surface with mouse or touch controls. The task of developing the user interface can often take up a considerable amount of the total effort spent on a project in computer science.

Touch screens are becoming popular for navigation systems. However, their logic differs only slightly from programming for mouse control. Due to this, you will now alter your city navigation program to support mouse control, so that the user can select the starting and destination point with a mouse click. Output information will be written both to the title bar and to the status bar.

The mouse click triggers an event that is processed in the callback *pressEvent()*. You register these in *makeGameGrid()* using the named parameter *mousePressed*. You should remember that the program can be in two different states depending on whether the user will click on the starting station as the next action, or whether they have already done so and have to choose the target station next. A Boolean variable (a flag) *isStart* will suffice for this state change, which will be *True* when the starting station has to be chosen next.

The program should be structured so that the user can perform the route search multiple times without having to restart the program. Therefore, the program has to internally reset to a well-defined initial state. This is referred to as **initialization** and is best carried out with the function *init()*. Since certain initializations are performed automatically at the start up, it is by no means trivial to repeatedly reset a currently running program to a well-defined initial state using its own function. Initialization errors are therefore programming errors which are widespread, dangerous, and difficult to locate since the program often behaves correctly during testing and only later behaves incorrectly when put into operation.




```
from gamegrid import *

locations = {
 'Althaus':Location(2, 0),
 'Bellevue':Location(0, 1),
 'City':Location(1, 3),
 'Dom':Location(4, 2),
 'Enge':Location(5, 0),
 'Friedhof':Location(3, 4)}

neighbours = {
 'Althaus':['Bellevue', 'Dom', 'Enge'],
```

```
 'Bellevue':['Althaus', 'City', 'Dom'],
 'City':['Bellevue', 'Dom', 'Friedhof'],
 'Dom':['Althaus', 'Bellevue', 'City', 'Enge', 'Friedhof'],
 'Enge':['Althaus', 'Dom'],
 'Friedhof':['Althaus', 'City', 'Dom']}

distances = {('Althaus', 'Bellevue'):5, ('Althaus', 'Dom'):9,
             ('Althaus', 'Enge'):6, ('Althaus', 'Friedhof'):15,
             ('Bellevue', 'City'):3, ('Bellevue', 'Dom'):13,
             ('City', 'Dom'):4, ('City', 'Friedhof'):3,
             ('Dom', 'Enge'):2, ('Dom', 'Friedhof'):12}

def getNeighbourDistance(station1, station2):
    if station1 < station2:
        return distances[(station1, station2)]
    return distances[(station2, station1)]

def totalDistance(li):
    sum = 0
    for i in range(len(li) - 1):
        sum += getNeighbourDistance(li[i], li[i + 1])
    return sum

def drawGraph():
    getBg().clear()
    getBg().setPaintColor(Color.blue)
    for station in locations:
        location = locations[station]
        getBg().fillCircle(toPoint(location), 10)
        startPoint = toPoint(location)
        getBg().drawText(station, startPoint)
        for s in neighbours[station]:
            drawConnection(station, s)
            if s < station:
                distance = distances[(s, station)]
            else:
                distance = distances[(station, s)]
            endPoint = toPoint(locations[s])
            getBg().drawText(str(distance),
                 getDividingPoint(startPoint, endPoint, 0.5))
    refresh()

def drawConnection(startStation, endStation):
    startPoint = toPoint(locations[startStation])
    endPoint = toPoint(locations[endStation])
    getBg().drawLine(startPoint, endPoint)

def search(station):
    global trackToTarget, trackLength
    visited.append(station)  # station marked as visited

    # Check for solution
    if station == targetStation:
        currentDistance = totalDistance(visited)
        if currentDistance < trackLength:
            trackLength = currentDistance
            trackToTarget = visited[:]

    for s in neighbours[station]:
        if s not in visited:  # if all are visited, recursion returns
            search(s) # recursive call
    visited.pop() # station may be visited by another path

def getStation(location):
    for station in locations:
        if locations[station] == location:
            return station
    return None # station not found
```

```
def init():
    global visited, trackToTarget, trackLength
    visited = []
    trackToTarget = []
    trackLength = 1000
    drawGraph()

def pressEvent(e):
    global isStart, startStation, targetStation
    mouseLoc = toLocationInGrid(e.getX(), e.getY())
    mouseStation = getStation(mouseLoc)
    if mouseStation == None:
        return
    if isStart:
        isStart = False
        init()
        setTitle("Click on destination  station")
        startStation = mouseStation
        getBg().setPaintColor(Color.red)
        getBg().fillCircle(toPoint(mouseLoc), 10)
    else:
        isStart = True
        setTitle("Once again? Click on  starting station.")
        targetStation = mouseStation
        getBg().setPaintColor(Color.green)
        getBg().fillCircle(toPoint(mouseLoc), 10)
        search(startStation)
        setStatusText("Shortest route from " + startStation + " to "
            + targetStation + ": " + str(trackToTarget) + " Length = "
            + str(trackLength))
        for i in range(len(trackToTarget) - 1):
            s1 = trackToTarget[i]
            s2 = trackToTarget[i + 1]
            getBg().setPaintColor(Color.black)
            getBg().setLineWidth(3)
            drawConnection(s1, s2)
            getBg().setLineWidth(1)
    refresh()

isStart = True
makeGameGrid(7, 5, 100, None, "sprites/city.png", False,
             mousePressed = pressEvent)
setTitle("City Guide. Click on starting  station.")
addStatusBar(30)
show()
init()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

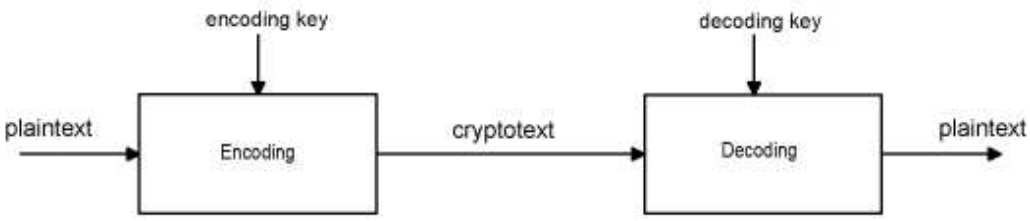
The algorithmic part with the backtracking remains pretty much unchanged. The user interface with mouse control is quite complex, despite good support from callbacks.

Using *global* easily leads to initialization errors in *Python*, as global variables can be created in functions and you might later forget to reset their value.

10.5 CRYPTOSYSTEMS

INTRODUCTION

The principle of secrecy plays an increasingly important role in our modern world. To protect privacy, but to also maintain confidentiality of important government, industrial, and military information, it is necessary to encrypt data so that, if they fell into the hands of the wrong people, it would be impossible or at least very difficult to find out the original information without the disclosure of the decryption method.
When **encoding** the original data, they are transformed into encoded data. During **decoding** the original data are restored. If the original data are written using the letter alphabet, we also speak of **plaintext** and **cryptotext**.
The description of the method used for decoding is called **key**. It can also simply consist of a single number, a number string, or a letter string (a keyword). If the same key is used for encoding and decoding, on also speaks of **symmetric-key cryptography**, or otherwise of an **asymmetric** (public-key) **cryptographic method**.



PROGRAMMING CONCEPTS: *Encoding, decoding, symmetric/asymmetric cryptography, Caesar cipher, Vigenère cipher, RSA encryption, private/public key*

CAESAR CIPHER

According to tradition, Julius Caesar (100 B.C. - 44 B.C.) already applied the following method for his military correspondence: Each plaintext letter is shifted down the alphabet to the right by a certain fixed number of places, continuing on to A again after Z.
With this method, the alphabet is thus laid out in a ring buffer. With the a shift (key) of 3, the letter A will be encoded in D, B in E, C in F, D in G, etc.

You use text files for data in your program so that you can change and share them easily. Write the plaintext using any text editor in the file *original.txt* that you have to save in the directory where your program is located. Only use capital letters and spaces for the text. You may write multiple lines, so for example:

```
TODAY WE MEET AT EIGHT
GREETINGS
TANIA
```

The encoder encodes the read text string msg from the file using the function *encode(msg)*, where each character is replaced by the corresponding crypto character, except for the line break \n.

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def encode(text):
    enc = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
            ch = alphabet[(i + key) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

Your encoded text reads as follows:

> XSHEBD IDQIIXDEXDIMKLX
> KVIIXMRKW
> XERME

The decoder is built analogously, except that the characters in the alphabet are shifted backwards.

```
import string
key = 4
alphabet = string.ascii_uppercase + " "

def decode(text):
    dec = ""
    for ch in text:
        if ch != "\n":
            i = alphabet.index(ch)
            ch = alphabet[(i - key) % 27]
        dec += ch
    return dec

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Keep in mind that you have to retain all empty spaces in the crypto text, even if they are at the beginning or the end of a line. It is clear that the encoding can be cracked easily. A good trick is to try it out for all key numbers 1 to 26.

ENCODING WITH THE VIGENÈRE METHOD

You can make Caesar cipher safer by applying a different alphabetic shift on each character of the plaintext. This so-called poly-alphabetic substitution could use any permutation of 27 numbers as a key. There is thus a huge number of possible keys, namely:

$27! = 10'888'869'450'418'352'160'768'000'000 \approx 10^{27}$

It is a bit easier to use a keyword that is assigned the list of the corresponding characters in the alphabet, so for example, the key ALICE is assigned to the list [0, 11, 8, 2, 4]. When encoding, the characters of the sequence are then shifted alphabetically to 0, 11, 8, 2, 4 and then repeated at 0, 11,... characters.

Blaise Vigenère (1523-1596)

```
import string
key = "ALICE"
alphabet = string.ascii_uppercase + " "

def encode(text):
    keyList = []
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i + keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

The decoder is again virtually identical.

```
import string
key = "ALICE"
alphabet = string.ascii_uppercase + " "

def decode(text):
    keyList = []
    for ch in key:
        i = alphabet.index(ch)
        keyList.append(i)
    print "keyList:", keyList
    enc = ""
    for n in range(len(text)):
        ch = text[n]
        if ch != "\n":
            i = alphabet.index(ch)
            k = n % len(key)
            ch = alphabet[(i - keyList[k]) % 27]
        enc += ch
    return enc

fInp = open("secret.txt")
krypto = fInp.read()
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

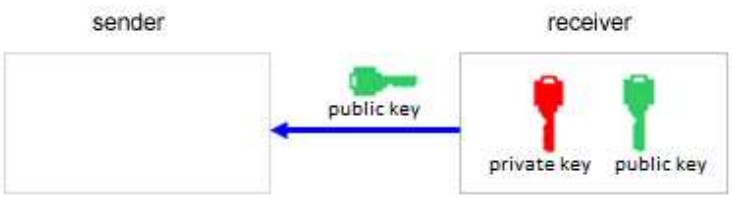
Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The Vigenère encoding method was invented in the 16th century by Blaise de Vigenère and was considered very safe for centuries. If someone knows that the length of the keyword is 5, they must nevertheless try out 26^5 = 11'881'376 key numbers unless they know something about the word itself, for example that it is the first name of a woman.

ENCODING WITH THE RSA METHOD

In this method, named after its inventors Rivest, Shamir and Adleman, a key-pair is used, namely a private key and a public key. The original data are encoded with the public key and decoded with the private key. It is an asymmetric cryptographic method.



Step 1:
The receiver generates the *private key* and the *public key* and sends the *public key* to the sender.

Step 2:
The sender encodes their message with the *public key* and sends the encoded text back.

Step 3:
The receiver decodes the text with the *private key*.

The keys are generated using the following algorithm based on number theory [**more...**].

First, two prime numbers p and q are chosen, which should have several hundred digits in order for the system to be secure. You multiply these and form m = p*q. From number theory, we know that the Euler function φ(m) = (p-1)*(q-1) is the total of co-prime numbers to m (a, b are co-prime if the largest common factor is ggT(a,b) = 1).

Next, you select a number e that is smaller than φ and co-prime to φ. With this, we already have the public key and it consists of the number pair:

Public key: [m, e]

Below is an example with the small prime numbers p = 73 and q = 151:

m = 73 * 151 = 11023, φ = 72 * 150 = 10800, e = 11 (chosen co-prime to φ)

Public key: [m, e] = [11023, 11]

The private key then consists of the number pair:

Private key: [m, d]

where for the number d we must ensure: (d * e) mod φ = 1

(since e and φ are co-prime, Bézout's lemma from number theory says that the equation has to have at least one solution).

You can determine the number d with your values for e and φ using a simple program by trying out 100,000 values for d in a for loop.

```
e = 11
phi = 10800

for d in range(100000):
    if  (d * e) % phi == 1:
        print "d", d
```

You get several solutions (5891, 16691, 27491, 49091, etc.). However, in principle you only need the first solution to determine the private key.

Private key: [m, d] = [11023, 5891]

In this case, calculating the private key is only so easy because you already know the numbers p and q, and hence also the number φ. Without knowing these numbers, the private key can only be calculated with great effort.

The RSA algorithm is used to encode numbers. In order to encode a text, you can use the ASCII code for each character and create the encoded value s with the public key [m, s] for the ciphertext according to the formula:

$s = r^e$ (modulo m).

Write these encoded numbers line by line in the file *secret.txt*.

```
publicKey = [11023, 11]

def encode(text):
```

```
    m = publicKey[0]
    e = publicKey[1]
    enc = ""
    for ch in text:
        r = ord(ch)
        s = int(r**e % m)
        enc += str(s) + "\n"
    return enc

fInp = open("original.txt")
text = fInp.read()
fInp.close()

print "Original:\n", text
krypto = encode(text)
print "Krypto:\n", krypto

fOut = open("secret.txt", "w")
for ch in krypto:
    fOut.write(ch)
fOut.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

In the decoder, you read the numbers from the file *secret.txt* (first) into a list. In order to decode, you calculate the original number according to the formula below which uses the private key s:

$r = s^d$ (modulo m).

This is the ASCII code of the original character.

```
privateKey = [11023, 5891]

def decode(li):
    m = privateKey[0]
    d = privateKey[1]
    enc = ""
    for c in li:
        s = int(c)
        r = s**d % m
        enc += chr(r)
    return enc

fInp = open("secret.txt")
krypto = []
while True:
   line = fInp.readline().rstrip("\n")
   if line == "":
       break
   krypto.append(line)
fInp.close()

print "Krypto:\n", krypto
msg = decode(krypto)
print "Message:\n", msg

fOut = open("message.txt", "w")
for ch in msg:
    fOut.write(ch)
fOut.close()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The big advantage of the RSA method is that no secret key information has to be exchanged between the sender and the receiver. Rather, the receiver generates both the public and the private key and only communicates the public key to the sender, while keeping the private key secret. The sender can now encode its data, but only the receiver can decode it [**more...**].

In practice, one could choose very large prime numbers p and q (several hundred digits longs). Generating the public key only requires the product formation m = p * q, which is very simple. If a hacker wants to find out the private key from the public key, they have to inversely determine both secret prime factors of m. Until now, factorizing a long number is only possible with an enormous amount of computational effort. That is how cryptosystems utilize the limits of calculability.

In principle, however, there are no absolutely secure encoding methods. Encoding is already considered to be safe once the time it takes to decode is considerably longer than the time during which the information is of importance.

EXERCISES

1. Try to decode the ciphertext
   OAL SHDXTKMJXSASTVVXHLSL XSAFNALTLAGF
   UMLSASVTFSGFDQSUXSL XJXSTLSXAZ L
   ZJXXLAFZK
   XNXDAFX

   It is a Caesar cipher.

   Note: One possible solution is to assume that the letter E occurs the most often in English texts. However, you can also try out all shifting possibilities.

2. Educate yourself on the Scytale encoding method from the Internet and implement an encoder/decoder based on its principle.

3. Explain why the Caesar cipher is a special case of the Vigenère method.

4. Generate a public and a private key using two prime numbers p and q (both smaller than 100) according to the RSA method and encode/decode a text.

10.6 FINITE-STATE MACHINES (AUTOMATA)

INTRODUCTION

If you want to investigate which problems a computer can generally solve, you first have to define what exactly a computing machine is. The famous mathematician and computer scientist Alan Turing published a study on this subject in 1936, long before a programmable digital computer even existed. The Turing machine, which was named after him, gradually runs programmatically through individual states. It does this on the basis of input values that it reads from a tape, and again writes output values onto the tape. This basic notion about how the computer functions are still valid today because each processor is actually a Turing machine that runs state by state at the ticks of a clock. However, state machines that can be modeled using transition graphs are better suited for practical applications. Because there is a finite number of states, they are called **finite-state machines**.

PROGRAMMING CONCEPTS: *Turing machine, finite-state machines (automata), Mealy machine, transition graph, formal language theory*

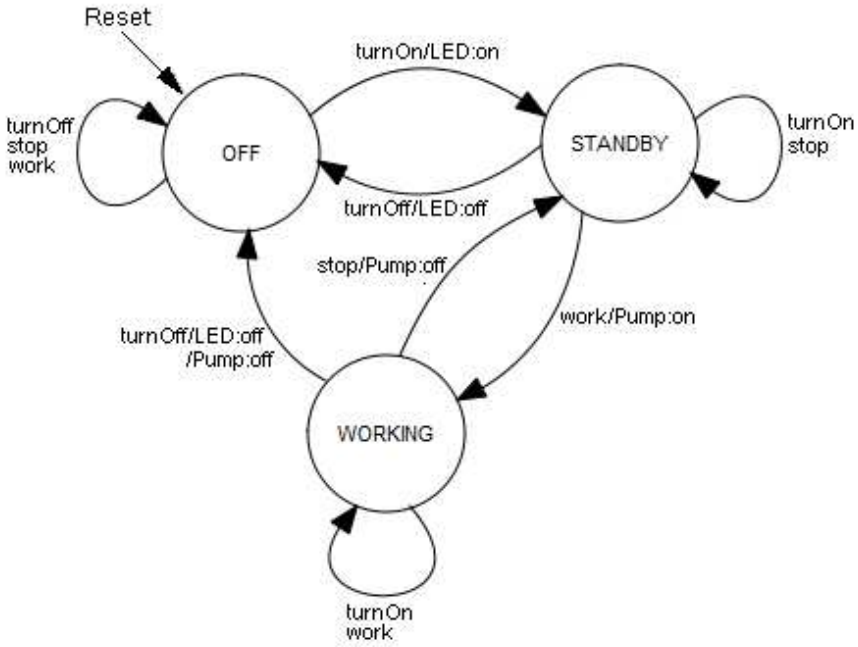
THE ESPRESSO MACHINE AS A MEALY MACHINE

You are bound to encounter many devices and machines that you can regard as automata every single day. Automata include devices such as vending machines, washing machines, ATMs, and many others. As an engineer and computer scientist you develop such machines with the clear understanding that these perform their task in such way that they move from the current state to a successor state, depending on the sensor data and the operation of the buttons and switches. You call these the **inputs** of the automaton. Depending on the input values, the automaton operates certain actuators at each transition, such as motors, pumps, lights, etc. These are the **outputs** of the automaton.

In this example, you will develop an espresso machine that has 3 states: It can be turned off (OFF), it can be enabled and ready for operation (STANDBY) and it can be actively pumping water to make an espresso (WORKING). There are 4 push buttons available that allow you to operate the machine: *turnOn*, *turnOff*, *work*, and *stop*.

Although you can describe the operation of an espresso machine in words, it becomes much clearer if you draw an transition graph. For this, you illustrate the states as circles and the transitions as arrows that you can denote with the inputs/outputs. It is also important to determine which initial state the automaton is in when you connect it to the power line. Since any key can be pressed in each state, all inputs have to be possible at every state. If no action is performed, the output is omitted.

Transition graph:



You can also record the behavior in a table in which you specify for each state *s* and each input *t* the successor state *s'*. You denote the initial state with a star.

Transition table:

t = s =	**OFF(*)**	**STANDBY**	**WORKING**
turnOff	OFF	OFF	OFF
turnOn	STANDBY	STANDBY	WORKING
stop	OFF	STANDBY	STANDBY
work	OFF	STANDBY	WORKING

In mathematical terms, you can say that the successor state s' is a function of the current state *s* and the input *t*: *s' = F(s, t)*. You call *F* the **transitional function**.

You can also record the outputs belonging to each state and input:

Output table:

t = s =	**OFF(*)**	**STANDBY**	**WORKING**
turnOff	-	LED off	LED off, Pump off
turnOn	LED on	-	-
stop	-	-	Pump off
work	-	Pump on	-

Here too, you can say mathematically that the output *g* is a function of the current state *s* and the input *t*: *g = G(s, t). G is called the* ***output function***.

MEMO

Together the states (with a labeled initial state), input values, and output values, as well as the transitional functions and output functions, form a so-called Mealy machine.

IMPLEMENTING THE ESPRESSO MACHINE WITH STRINGS

A key press should trigger the transition from one state to the next. The respective key, one of the 4 cursor keys, specifies the input value. The implementation is quite straightforward: The program waits on a key entry in an endless event loop with *getKeyEvent()*. With the return value, the current state is changed according to the transition table and the outputs are given according to the output table.

```
from gconsole import *

def getKeyEvent():
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_UP:
        return "stop"
    if keyCode == KeyEvent.VK_DOWN:
        return "work"
    if keyCode == KeyEvent.VK_LEFT:
        return "turnOff"
    if keyCode == KeyEvent.VK_RIGHT:
        return "turnOn"
    return ""

state = "OFF"  # Start state
makeConsole()
while True:
    gprintln("State: " + state)
    entry = getKeyEvent()
    if entry == "turnOff":
       if state == "STANDBY":
            state = "OFF"
            gprintln("LED off")
       if state == "WORKING":
            state = "OFF"
            gprintln("LED and pump off")
    elif entry == "turnOn":
        if state == "OFF":
            state = "STANDBY"
            gprintln("LED enabled")
    elif entry == "stop":
        if state == "WORKING":
            state = "STANDBY"
            gprintln("Pumpe off")
    elif entry == "work":
        if state == "STANDBY":
            state = "WORKING"
            gprintln("Pumpe enabled")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Only the events that lead to a change of state or generate an output are treated in the event loop.

ENUMERATIONS AS STATE AND EVENT IDENTIFIERS

Since the automaton operates with certain states, input values, and output values, it makes sense to introduce a particular data structure for that. Many programming languages offer a specific data type for enumeration. Unfortunately this data type is missing in the standard syntax of *Python,* but it can be added in *TigerJython* using the additional keyword *enum()*. You should use strings when defining the enumeration values, however they must adhere to the variable naming convention.

```
from gconsole import *

def getKeyEvent():
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_UP:
        return Events.stop
    if keyCode == KeyEvent.VK_DOWN:
        return Events.work
    if keyCode == KeyEvent.VK_LEFT:
        return Events.turnOff
    if keyCode == KeyEvent.VK_RIGHT:
        return Events.turnOn
    return None

State = enum("OFF", "STANDBY", "WORKING")
state = State.OFF
Events = enum("turnOn", "turnOff", "stop", "work")
makeConsole()
while True:
    gprintln("State: " + str(state))
    entry = getKeyEvent()
    if entry == Events.turnOn:
        if state == State.OFF:
            state = State.STANDBY
    elif entry == Events.turnOff:
            state = State.OFF
    elif entry == Events.work:
        if state == State.STANDBY:
            state = State.WORKING
    elif entry == Events.stop:
        if state == State.WORKING:
            state = State.STANDBY
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

It is up to you whether you want to use the additional data type *enum*. The programs will not be shorter, but rather more clear and secure, since only enumeration values defined in the *enum* may occur.

MOUSE-CONTROLLED IMPLEMENTATION OF ESPRESSO MACHINE

With just a little extra effort you can simulate the espresso machine graphically using the *JGameGrid* library, whereby the program gains a lot of clarity and programming becomes a lot more fun. Instead of with the keyboard, the 4 inputs are triggered by mouse clicks on simulated buttons, and the output of the LED and the pump are made immediately visible with sprite images. Instead of the event loop, you use the callback *pressEvent()* which will always be called when you click on the image with the mouse. Since you use a grid with 7 x 11 cells as a *GameGrid*, you can capture the clicks on the buttons using grid coordinates.

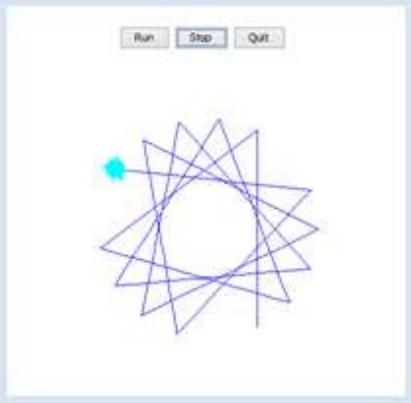
```
from gamegrid import *

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc == Location(1, 2): # off
        state = State.OFF
        led.show(0)
        coffee.hide()
    elif loc == Location(2, 2): # on
        if state == State.OFF:
            state = State.STANDBY
            led.show(1)
    elif loc == Location(4, 2): # stop
        if state == State.WORKING:
            state = State.STANDBY
            coffee.hide()
    elif loc == Location(5, 2): # work
        if state == State.STANDBY:
            state = State.WORKING
            coffee.show()
    setTitle("State: " + str(state))
    refresh()

State = enum("OFF", "STANDBY", "WORKING")
state = State.OFF
makeGameGrid(7, 11, 50, None, "sprites/espresso.png", False,
             mousePressed = pressEvent)
show()
setTitle("State: " + str(state))
led = Actor("sprites/lightout.gif", 2)
addActor(led, Location(3, 3))
coffee = Actor("sprites/coffee.png")
addActor(coffee, Location(3, 6))
coffee.hide()
refresh()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

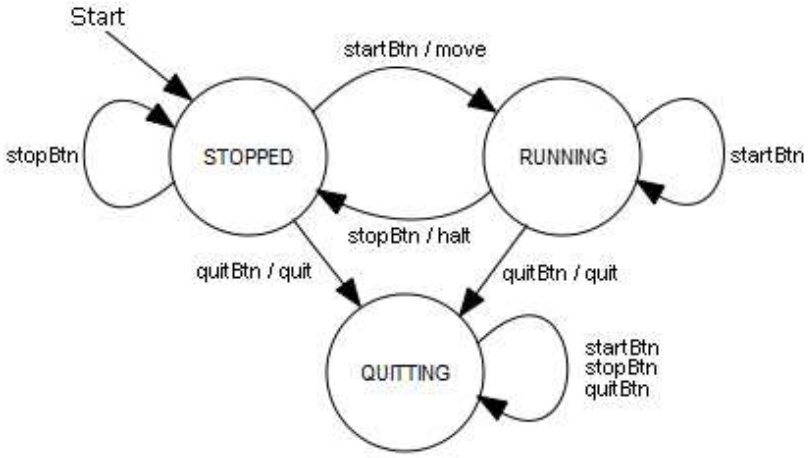
MEMO

A simulation becomes very clear and attractive with a graphical user interface.

THINKING IN STATES WITH GRAPHICAL USER INTERFACES



At first glance, Mealy machines appear to be a rather theoretical matter. This is not at all the case. Rather, you must always think in states with modern, event-driven programs with a graphical user interface. As an example, you write a turtle program controlled by 3 buttons: The *start_button* sets the turtle in motion, the *stop_button* stops the movement, and the *quit_button* ends the program. In order to correctly implement the program, you have to keep the transition graph in mind::



As you already know, no animations and only short-lasting code may be executed in GUI event callbacks, since the screen is only re-rendered at the end of the function. This is why in the callbacks of the button clicks you only shift the state, and you execute the movement of the turtle in the main part of the program.

(You can learn more about this problem in Appendix 4: **Parallel processing**)

```
from javax.swing import JButton
from gturtle import *

def buttonCallback(evt):
    global state
    source = evt.getSource()
    if source == runBtn:
        state = State.RUNNING
        setTitle("State: RUNNING")
    if source == stopBtn:
        state = State.STOPPED
        setTitle("State: STOPPED")
    if source == quitBtn:
        state = State.QUITTING
        setTitle("State: QUITTING")

State = enum("STOPPED", "RUNNING", "QUITTING")
state = State.STOPPED

runBtn = JButton("Run", actionPerformed = buttonCallback)
stopBtn = JButton("Stop", actionPerformed = buttonCallback)
quitBtn = JButton("Quit", actionPerformed = buttonCallback)
makeTurtle()
setTitle("State: STOPPED")
back(100)

pg = getPlayground()
pg.add(runBtn)
pg.add(stopBtn)
pg.add(quitBtn)
pg.validate()

while state != State.QUITTING and not isDisposed():
    if state == State.RUNNING:
          forward(200).left(127)
dispose()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

This program structure is typical of event-driven programs and you should try to remember it well. You have to import the package *JButton* in order to use the buttons, and then add them to the turtle window (the playground) using *add()*. You have to re-render the turtle window using *validate()* in order to make them visible.

EXERCISES

1. A parking ticket vending machine only takes 1 € and 2 € coins that are inserted individually, one after the other. Once the machine has received at least the amount of the parking fee, it issues the ticket and gives back the change if you overpaid. The parking fee is 3 €.

   Starting at the initial state S0, the machine runs through the states S1 or S2, depending on whether you inserted a 1 € or a 2 € coin. It outputs the values - (nothing), K (ticket), or K,R (ticket and change).

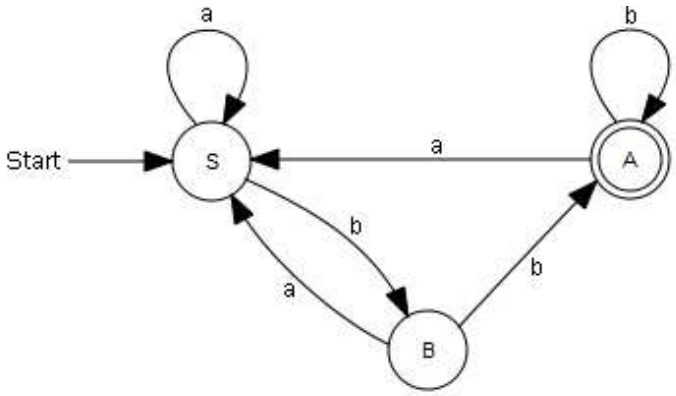
   a. Create the input and output tables
   b. Draw the transition graph
   c. Create a program using the *GConsole*, which interprets the pressing of the number 1 key as an insertion of a 1 € coin and the number 2 as a 2 € coin and writes the subsequent state and the output values to the console window.

ADDITIONAL MATERIAL

ACCEPTORS FOR REGULAR LANGUAGES

A formal language consists of an alphabet of symbols and a set of rules, which allows someone to clearly decide whether a particular sequence of symbols belongs to the language. If it is possible to implement the set of rules using an automaton, it is called a **regular language**.

As an example, look at a very simple language with an alphabet consisting only of the letters a and b. You can regard the set of rules as a special case of a Mealy machine that does not generate output values. In this case, the machine reads character by character starting at an initial state and transitions to a successive state based on the read character. If it is located in one of the predetermined final states after reading the last character, the word does indeed belong to the language. Look at the following transition graph (S: initial state, A: end state):



In the implementation, the change of state is triggered by pressing the character keys a or b. Then, your program writes out the current state and the input word.

```
from gconsole import *

def getKeyEvent():
    global word
    keyCode = getKeyCodeWait(True)
    if keyCode == KeyEvent.VK_A:
        return Events.a
    if keyCode == KeyEvent.VK_B:
        return Events.b
    return None

State = enum("S", "A", "B")
state = State.S
Events = enum("a", "b")
makeConsole()
word = ""
gprintln("State: " + str(state))
while True:
    entry = getKeyEvent()
    if entry == Events.a:
        if state == State.A:
            state = State.S
        elif state == State.B:
            state = State.S
        word += "a"
        gprint("Word: " + word + " -> ")
        gprintln("State: " + str(state))
    elif entry == Events.b:
        if state == State.S:
            state = State.B
        elif state == State.B:
            state = State.A
        word += "b"
        gprint("Word: " + word + " -> ")
        gprintln("State: " + str(state))
```

MEMO

An acceptor checks to see if a word belongs to a language. It is a special case of a Mealy machine without output values. The word does indeed belong to the language if you arrive at the end state A after reading all letters, beginning in the initial state S. For example, *abbabb* belongs to the language, whereas *baabaa* does not.

EXERCISES

1. A laughing machine should accept only the words *ha.* or *haha.* or *hahaha. etc*. (last character a period). Draw the transition graph and implement it.

   Additional instructions: You can introduce an error state E to the program, which cannot be left anymore with any input.

10.7 INFORMATION & ORDER

INTRODUCTION

Even though the word *information* is used often, it is not that easy to grasp the concept precisely and to make it measurable. In daily life, more information is associated with more knowledge and it is said that a person A informed in a certain thing knows more than an uninformed person B.

In order to measure how much more information A has than B (or team B), or the lack of information of B with respect to A, you best imagine a TV quiz show. Person or team B has to find something out that only person A knows, for example their profession. Thereby B asks questions that A has to answer with a Yes or a No. The rules are as follows:

The lack of information I of B in relation to A (in bits) is the number of questions with Yes/No answers that B (on average and at an optimal asking strategy) has to ask in order to have the same knowledge about a certain thing as A has.

PROGRAMMING CONCEPTS: *Information, information content, entropy*

NUMBER GUESSING GAME

You can enact such a guessing game in your classroom (or just in your head) as follows: Your classmate Judith is sent out of the room. One of the remaining W = 16 classmates receives a "desired" object, such as a bar of chocolate. After Judith is called back into the classroom, you and your classmates know who has the chocolate, but Judith does not. How big is her lack of information?

For the sake of simplicity, the classmates are numbered from 0 to 15 and Judith can now ask everyone questions in order to find out the secret number of the classmate with the chocolate. The less questions she asks, the better. One possibility would be for her to ask any random number: "Is it 13?". If the answer is No, she asks a different number.

Judith could also proceed systematically and ask starting from 0: "Is it 0?", then "Is it 1?", etc. In your computer simulation, you determine how many questions she needs to ask **on average** (i.e. if the game is played many times) with this type of questions, in order to finally find out the secret number. Think about the following:

If the secret number is 0, then 1 question is needed to find the answer; if the secret number is 1, 2 questions are needed, etc. If the secret number is 14, then 15 questions are needed, but when the secret number is 15, again only 15 questions are needed since it is the last possible option. The program plays the game 100,000 times and counts the number of questions it takes in each case until the secret number is determined. Those numbers are added up to finally determine the average.

```
import random

sum = 0
z = 100000
repeat z:
    n = random.randint(0, 15)
    if n != 15:
```

```
            q = n + 1 # number of questions
        else:
            q = 15
        sum += q
    print "Mean:", sum / z
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

With this strategy, Judith would need to ask an average of 8.43 questions. That is quite a lot. But since Judith is clever, she decides to use a much better strategy. She asks them: *"Is it a number between 0 and 7?"* (limits included). If the answer is *Yes*, she divides the area into two equal parts again and asks: *"Is the number between 0 and 3?".* If the answer is again *Yes,* she asks *"Is it 0 or 1?"* and if the answer is *Yes* she asks: *"Is it 0?".* No matter what the response, she now knows the number. With this "binary" (questioning) strategy and when W = 16, Judith always has to ask exactly 4 questions and therefore 4 is the average number of questions asked. This shows us that the binary (questioning) strategy is optimal and the information about the owner of the chocolate bar has thus the value I = 4 bit.

MEMO

As you can easily find out, with W = 32 numbers the information amounts to I = 5 bit, and with W = 64 numbers I = 6 bit. So apparently we get $2^I = W$ or $I = ld(W)$. It also does not need to be a number that one is searching for, but rather it can consist of any one state that has to be found out from *W* (equally probable) number of states.

So the **information** at the knowledge of a state from W equally probable states is

$$I = ld(W)$$ (ld is the logarithm dualis, logarithm with base 2)

INFORMATION CONTENT OF A WORD

Since the W states are equally likely, the probability of the states is p = 1/W. You can also write the information as follows:

$$I = ld(1/p) = -ld(p)$$

You certainly know that the probabilities of certain letters in a spoken language differ greatly. The following table shows the probabilities for the English language [**more...**].

A	8.34%	I	6.71%	Q	0.09%	Y	2.04%
B	1.54%	J	0.23%	R	5.68%	Z	0.96%
C	2.73%	K	0.87%	S	6.11%		
D	4.14%	L	4.24%	T	9.37%		
E	12.60%	M	2.53%	U	2.85%		
F	2.03%	N	6.80%	V	1.06%		
G	1.92%	O	7.70%	W	2.34%		
H	6.11%	P	1.66%	X	0.20%		

It is certainly not the case that the probability of a letter in a word is independent of the letters of the words you already know and the context in which the word appears. However, if we make this simple assumption, then the probability p of a 2-letter combination with the individual probabilities p1 and p2 is, according to the product rule, p = p1 * p2 and for the information:

$$I = -ld(p) = -ld(p_1 * p_2) = -ld(p_1) - ld(p_2)$$

or for arbitrarily many letters:

$$I = -ld(p_1) - ld(p_2) - ld(p_3) - \ldots - ld(p_n) = -\Sigma\, ld(p_i)$$

In your program, you can enter a word and you will get the information content of the word as an output. For this you should download the files with the letter frequencies in German, English, and French from **here** and copy them to the directory where your *Python* program is.

```
import math

f = open("letterfreq_de.txt")
s = "{"
for line in f:
    line = line[:-1] # remove trailing \n
    s += line + ", "
f.close()
s = s[:-2]  # remove trailing ,
s += "}"
occurance = eval(s)

while True:
    word = inputString("Enter a word")
    I = 0
    for letter in word.upper():
        p = occurance[letter] / 100
        I -= math.log(p, 2)
    print word, "-> I =", round(I, 2), "bit"
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

The data are saved in the text file line by line in the format 'letter' : percentage. In *Python*, a dictionary is well suited as a data structure for this. In order to convert the data information to a dictionary, the lines are read and packed into a string in standard dictionary format {key : value, key : value,...}. This string is interpreted as a line of code with *eval()* and a dictionary is created.

As you can find out with your program, the information content of words with rare letters is greater. However, the information content determined this way has nothing to do with the importance or personal relevance of a certain word in a certain context, or in other words whether the information connected to the word is trivial for you or of crucial importance. It is far beyond the capabilities of today's information systems to determine a measure for this.

EXERCISES

1. When playing a guessing game with 16 classmates, the clever classmate could also ask something else by saying that you should imagine the secret number of the person with the chocolate as a binary number with 4 digits. Write out how you would ask this.

2. Use the list of English words from the file *words-1$.txt* and determine the word with the smallest and largest information content from words that have a length of 5. Comment on your findings. Download the word lists from **here**.

3*. Determine the average number of questions it takes in the guessing of the number, in a situation where the numbers go from 0 to 15 and you use the questioning strategy "Is it 0, is it 1, etc." from a theoretical consideration..

ADDITIONAL MATERIAL

THE RELATIONSHIP BETWEEN DISORDER AND ENTROPY

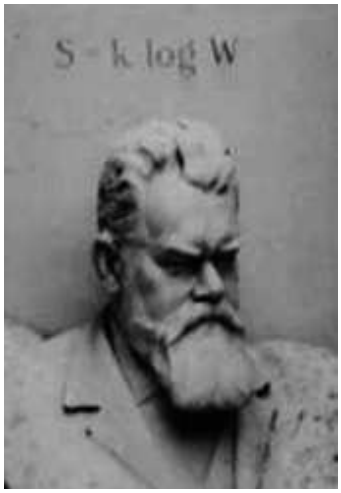
There is an extremely interesting relationship between the information that we have about a system and the order of the system. For example, the order is certainly much greater when all students in a classroom are sitting at their desks, than when they are all moving around randomly in the room. In the disordered state, our lack of information of where each individual person is located is very high, thus we can use this lack of information as a measure of disorder. For this, one defines the concept of entropy:

Entropy (in bits) is the lack of information I which we possess when describing the system macroscopically, in relation to someone who knows the microscopic state. For the entropy in (J/K) we set $S = k * ln2 * I$

The factor k is the Bolzmann constant. Entropy and the lack of information are thus the same except for one pre-factor. If we assume a system with W equally probable states, we have:

$S = k * ln2 * ldW$ or **$S = k * \log W$**

This fundamental relationship comes from famous physicist Ludwig Boltzmann (1844-1900) and it is written on his gravestone.



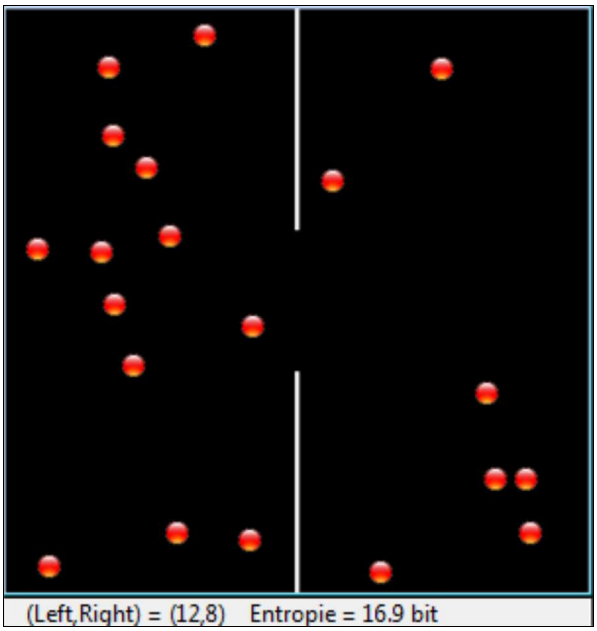
As you can determine, systems left to their own resources have the tendency to move from an ordered to a disordered state. For example:

- Passengers in a train spread out over the entire car
- Cigarette smoke spreads in a room
- A blot of ink disperses in a water glass
- The temperature between the coffee and the cup evens out over time

Since the disordered state has a higher entropy than the ordered state, one can formulate this as a law of nature (2nd law of thermodynamics):

In a closed system, the entropy either increases or remains the same, but it never decreases.

You simulate a particle system with your program where the particles act as gas atoms and collide with each other while exchanging their direction and kinetic energy. All of the particles should first be located in the left part of the container, where they can escape through a hole in the divider. What happens?



The module *gamegrid* is used for the animation. The particles are modeled in the class *Particle* that is derived from the class *Actor.* You can move the particles using the method *act()*, which is automatically called in each simulation cycle. The collisions are handled with collision events. For this, derive the class *CollisionListener* from *GGActorCollisionListener* and override the method *collide()*. This procedure is the same as what you saw in chapter 8.10 on **Brownian Movement**. The 20 particles are then divided into 4 different velocity groups. Since the velocities are

switched between particles that collide, the total energy of the system remains constant, i.e. the system is closed.

```
from gamegrid import *
from gpanel import *
import math
import random

# ================== class Particle ====================
class Particle(Actor):
    def __init__(self):
        Actor.__init__(self, "sprites/ball_0.gif")

    # Called when actor is added to gamegrid
    def reset(self):
        self.isLeft = True

    def advance(self, distance):
        pt = self.gameGrid.toPoint(self.getNextMoveLocation())
        dir = self.getDirection()
        # Left/right wall
        if pt.x < 0 or pt.x > w:
            self.setDirection(180 - dir)
        # Top/bottom wall
        if pt.y < 0 or pt.y > h:
            self.setDirection(360 - dir)
        # Separation
        if (pt.y < h // 2 - r or pt.y > h // 2 + r) and \
            pt.x > self.gameGrid.getPgWidth() // 2 - 2 and \
            pt.x < self.gameGrid.getPgWidth() // 2 + 2:
                self.setDirection(180 - dir)

        self.move(distance)
        if self.getX() < w // 2:
            self.isLeft = True
        else:
            self.isLeft = False

    def act(self):
        self.advance(3)

    def atLeft(self):
        return self.isLeft

# ================== class CollisionListener =========
class CollisionListener(GGActorCollisionListener):
    # Collision callback: just exchange direction and speed
    def collide(self, a,  b):
        dir1 = a.getDirection()
        dir2 = b.getDirection()
        sd1 = a.getSlowDown()
        sd2 = b.getSlowDown()
        a.setDirection(dir2)
        a.setSlowDown(sd2)
        b.setDirection(dir1)
        b.setSlowDown(sd1)
        return 5  # Wait a moment until collision is rearmed


# ================== Global sections =================
def drawSeparation():
    getBg().setLineWidth(3)
    getBg().drawLine(w // 2, 0, w // 2, h // 2 - r)
    getBg().drawLine(w // 2, h, w // 2, h // 2 + r)

def init():
    collisionListener = CollisionListener()
    for i in range(nbParticles):
```

```
        particles[i] = Particle()

        # Put them at random locations, but apart of each other
        ok = False
        while not ok:
            ok = True
            loc = getRandomLocation()
            if loc.x > w / 2 - 20:
                ok = False
            continue

        for k in range(i):
            dx = particles[k].getLocation().x - loc.x
            dy = particles[k].getLocation().y - loc.y
            if dx * dx + dy * dy < 300:
                ok = False
        addActor(particles[i], loc, getRandomDirection())
        delay(100)

        # Select collision area
        particles[i].setCollisionCircle(Point(0, 0), 8)
        # Select collision listener
        particles[i].addActorCollisionListener(collisionListener)

        # Set speed in groups of 5
        if i < 5:
            particles[i].setSlowDown(2)
        elif i < 10:
            particles[i].setSlowDown(3)
        elif i < 15:
            particles[i].setSlowDown(4)

    # Define collision partners
    for i in range(nbParticles):
        for k in range(i + 1, nbParticles):
            particles[i].addCollisionActor(particles[k])

def binomial(n, k):
    if k < 0 or k > n:
        return 0
    if k == 0 or k == n:
        return 1
    k = min(k, n - k) # take advantage of symmetry
    c = 1
    for i in range(k):
        c = c * (n - i) / (i + 1)
    return c

r = 50  # Radius of hole
w = 400
h = 400
nbParticles = 20
particles = [0] * nbParticles
makeGPanel(Size(600, 300))
window(-6, 66, -2, 22)
title("Entropy")
windowPosition(600, 20)
drawGrid(0, 60, 0, 20)
makeGameGrid(w, h, 1, False)
setSimulationPeriod(10)
addStatusBar(20)
drawSeparation()
setTitle("Entropy")
show()
init()
doRun()

t = 0
while not isDisposed():
```

```
    nbLeft = 0
    for particle in particles:
        if particle.atLeft():
            nbLeft += 1
    entropy = round(math.log(binomial(nbParticles, nbLeft), 2), 1)
    setStatusText("(Left,Right) = (" + str(nbLeft) +
           "," + str(nbParticles - nbLeft) + ")" +
           "    Entropie = " + str(entropy) + " bit")
    if t % 60 == 0:
        clear()
        lineWidth(1)
        drawGrid(0, 60, 0, 20)
        lineWidth(3)
        move(0, entropy)
    else:
        draw(t % 60, entropy)
    t += 1
    delay(1000)
dispose() # GPanel
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

When seen from the outside (macroscopically), a state is determined by the number k of particles in the right part, and thus N - k in the left part. A macroscopic observation lacks the precise knowledge, which k of the N numbered particles are on the right. According to combinatorics there are

$$W = \binom{N}{k} = \frac{N!}{k! \cdot (N-k)!}$$

such possibilities. The lack of information is therefore

$$I = ld(W) = ld\binom{N}{k}$$ and the entropy $$S = \log(w) = k \cdot log\binom{N}{k}$$

The temporal process is plotted in a *GPanel* graphic. You can clearly see that the particles distribute themselves throughout the entire container over time, but there is also a chance that all particles end up again in the left half. The probability of this decreases rapidly with an increasing number of particles. The 2nd law thus relies on a statistical property of multiple-particle systems.

Since the reverse process is not observed, these processes are called **irreversible**. However, there are indeed processes that passed from disorder to order. For this, they need an "ordering constraint" [**more...**].

Some examples are:

- The matter of the cosmos forms galaxies, stars, and planetary systems
- Life emerges from non-living matter
- With sufficient enforcement the disorder in kitchens, or classrooms, diminishes
- Clouds (liquid) and ice (solid) emerge from the cooling of water vapor (gas). Thus, phase transitions change the order
- Concert-goers flock back to their seats after the break

These processes decrease the disorder, and therefore the entropy, so that structures gradually become visible from chaos.

EXERCISES

1. Modify the gas simulation so that after 50 seconds a "demon" makes sure that the particles only fly from right to left through the gap, but not from left to right. Show that entropy now decreases.

2. Find other examples of systems that:

   a. switch from order to disorder

   b. switch from disorder to order. State the ordering constraint.

chapter eleven

APPENDIX

Learning Objectives

- You expand your knowledge of algorithms and their implementation in *Python*.
- You know in which contexts programming errors are particularly often made.
- You know important procedures for troubleshooting (debugging).
- You know what parallel processing means and you can write simple programs with their own threads.
- You can describe what is meant by race conditions and deadlock.

11.1 FUN MIND GAMES

INTRODUCTION

Mind games with mathematical backgrounds are very popular and wide-spread. The goal of this chapter is not to spoil your pleasure of solving such puzzles "by hand" using paper and pen. Much rather, you should experience that the computer can find the solution by solving systematically using backtracking, however, with two significant differences. On the one hand, if no other limitations and strategies are incorporated into the solution process, the solution can take an enormous amount of time, even on a fast computer. On the other hand, the computer can basically find all solutions which may be very difficult to do by hand. This makes it possible to use the computer to provide evidence that a certain solution is the simplest (shortest).

SUDOKU

The game Sudoku has boomed and spread ever since 1980, and today it is very popular. You can find them in the puzzle section of many different daily and weekly newspapers. They work like a typical number puzzle with simple rules. In the standard version, the numbers 1 to 9 must be placed in a 9x9 grid so that every number appears exactly once in each row and each column. In addition, the grid is divided into 9 sub-grids with 3x3 cells where the numbers have to occur exactly once. At the beginning of the game, a certain number of cells are already occupied. With an ideal initial situation there should be only one solution.

Depending on the initial situation, the puzzle can be easier or more difficult to solve. An experienced Sudoku player uses certain well-known or personal strategies to solve the game. When backtracking with the computer, open cells are systematically one by one filled with numbers from 1 to 9, so that there is no conflict with the rules of the game. If there is no possibility anymore, the last turn is cancelled.

In the following, the computer uses this backtracking algorithm. Using a *GameGrid* is ideal for the graphical representation, since the game has a grid structure. You draw the initial numbers in black as a *TextActor*, and the inserted numbers in red.

As you know about backtracking from **chapter 10.3.** you must first find a favorable data structure for the game states. Since there is a 9x9 grid in this case, you should choose a list of 9 row lists. The number 0 indicates an empty cell. Thus, the game shown here begins with the following state:

```
startState = [ \
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]
```

As usual, you will develop the program step by step. Your first task is to display the game state in the *GameGrid* and give the user the ability to assign the empty cells with a mouse click. While this is unnecessary in the automatic search of the solution, it allows you to interactively influence the game, which can be especially helpful in the testing phase. Also, this way you can solve the puzzle on the screen instead of on a piece of paper.

```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    x = loc.x
    y = loc.y
    value = startState[y][x]
    value = (value + 1)  % 10
    startState[y][x] = value
    showState(startState)

def showState(state):
    removeAllActors()
    for y in range(9):
        for x in range(9):
            loc = Location(x, y)
            value = state[y][x]
            if  value != 0:
                if loc in fixedLocations:
                    c = Color.black
                else:
                    c = Color.red
                digit = TextActor(str(value), c, Color.white,
                        Font("Arial", Font.BOLD, 20))
                addActorNoRefresh(digit, loc)
    refresh()

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

startState = [
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]

fixedLocations = []
for x in range(9):
    for y in range(9):
        if startState[y][x] != 0:
            fixedLocations.append(Location(x, y))

showState(startState)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

The next step is to build a function *isValid(state)* that checks if a given game state *state* complies with the rules of the game. This is tedious work because you have to check the 9 rows and 9 columns, as well as the 9 square blocks. You can write out the result to the status bar.


```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    xs = loc.x // 3
    ys = loc.y // 3
    x = loc.x % 3
    y = loc.y % 3
    value = startState[ys][xs][y][x]
    value = (value + 1)  % 10
    startState[ys][xs][y][x] = value
    showState(startState)
    if isValid(startState):
        setStatusText("State valid")
    else:
        setStatusText("Invalid state")

def showState(state):
    removeAllActors()
    for ys in range(3):
        for xs in range(3):
            for y in range(3):
                for x in range(3):
                    loc = Location(x + 3 * xs, y + 3 * ys)
                    value = state[ys][xs][y][x]
                    if  value != 0:
                        if loc in fixedLocations:
                            c = Color.black
                        else:
                            c = Color.red
                        digit = TextActor(str(value), c, Color.white,
                                Font("Arial", Font.BOLD, 20))
                        addActorNoRefresh(digit, loc)
    refresh()

def isValid(state):
    # Check lines
    for ys in range(3):
        for y in range(3):
            line = []
            for xs in range(3):
                for x in range(3):
                    value = state[ys][xs][y][x]
                    if value > 0 and value in line:
                        return False
                    else:
                        line.append(value)
```

```
    # Check rows
    for xs in range(3):
        for x in range(3):
            row = []
            for ys in range(3):
                for y in range(3):
                    value = state[ys][xs][y][x]
                    if value > 0 and value in row:
                        return False
                    else:
                        row.append(value)

    # Check subgrids
    for ys in range(3):
        for xs in range(3):
            subgrid = state[ys][xs]
            square = []
            for y in range(3):
                for x in range(3):
                    value = subgrid[y][x]
                    if value > 0 and value in square:
                        return False
                    else:
                        square.append(value)
    return True

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
addStatusBar(30)
visited = []
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

stateWiki = [[[[0, 3, 0], [0, 0, 0], [0, 0, 8]],
              [[0, 0, 0], [1, 9, 5], [0, 0, 0]],
              [[0, 0, 0], [0, 0, 0], [0, 6, 0]]],
             [[[8, 0, 0], [4, 0, 0], [0, 0, 0]],
              [[0, 6, 0], [8, 0, 0], [0, 2, 0]],
              [[0, 0, 0], [0, 0, 1], [0, 0, 0]]],
             [[[0, 6, 0], [0, 0, 0], [0, 0, 0]],
              [[0, 0, 0], [4, 1, 9], [0, 0, 0]],
              [[2, 8, 0], [0, 0, 5], [0, 7, 0]]]]
startState = stateWiki

fixedLocations = []
for xs in range(3):
    for ys in range(3):
        for x in range(3):
            for y in range(3):
                if startState[ys][xs][y][x] != 0:
                    fixedLocations.append(Location(x + 3 * xs, y + 3 * ys))

showState(startState)
```

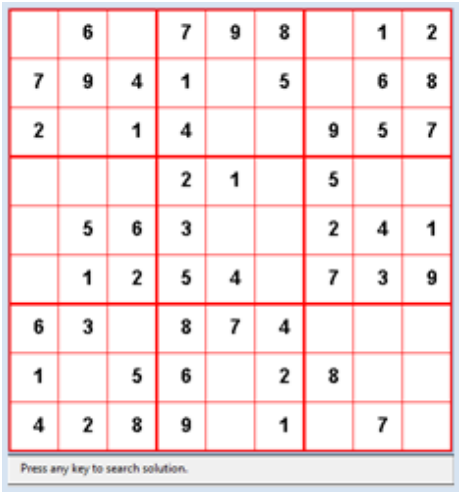
Highlight program code (Ctrl+C copy, Ctrl+V paste)

For the backtracking, you have to determine the subsequent nodes of a state with the function *getNeighbours(state)*. For this, you choose an empty cell with *getEmptyCell()* and insert all numbers in order which belong to a legal game state. At least one of them will be the correct one.

In *getNeighbours(state),* you first copy the given list of states into a clone, since you are not

allowed to overwrite the state received as a parameter. Then, you fill the empty cells successively with the numbers 1 to 9 and add copies of the allowed states into the neighbor list, which you then finally return [**more...**]

You can adopt the recursively defined function *search(),* which performs the actual backtracking, almost without changing anything from other backtracking problems. In this case, you are looking for just a single solution and you end the recursive call with the flag *found*.




```
from gamegrid import *

def pressEvent(e):
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in fixedLocations:
        setStatusText("Location fixed")
        return
    x = loc.x
    y = loc.y
    value = startState[y][x]
    value = (value + 1)  % 10
    startState[y][x] = value
    showState(startState)
    if isValid(startState):
        setStatusText("State valid")
    else:
        setStatusText("Invalid state")

def getBlockValues(state, x, y):
    return [state[y][x], state[y][x + 1], state[y][x + 2],
            state[y + 1][x], state[y + 1][x + 1], state[y + 1][x + 2],
            state[y + 2][x], state[y + 2][x + 1], state[ y + 2][x + 2]]

def showState(state):
    removeAllActors()
    for y in range(9):
        for x in range(9):
            loc = Location(x, y)
            value = state[y][x]
            if  value != 0:
                if loc in fixedLocations:
                    c = Color.black
                else:
                    c = Color.red
                digit = TextActor(str(value), c, Color.white,
                        Font("Arial", Font.BOLD, 20))
                addActorNoRefresh(digit, loc)
    refresh()

def isValid(state):
    # Check lines
```

```
    for y in range(9):
        values = []
        for x in range(9):
            value = state[y][x]
            if value > 0 and value in values:
                return False
            else:
                values.append(value)
    # Check rows
    for x in range(9):
        values = []
        for y in range(9):
            value = state[y][x]
            if value > 0 and value in values:
                return False
            else:
                values.append(value)

    # Check blocks
    for yblock in range(3):
        for xblock in range(3):
            values = []
            li = getBlockValues(state, 3 * xblock, 3 * yblock)
            for value in li:
                if value > 0 and value in values:
                    return False
                else:
                    values.append(value)
    return True

def getEmptyCell(state):
    emptyCells = []
    for y in range(9):
        for x in range(9):
            if state[y][x] == 0:
                return [x, y]
    return []

def cloneState(state):
    li = []
    for y in range(9):
        line = []
        for x in range(9):
           line.append(state[y][x])
        li.append(line)
    return li

def getNeighbours(state):
    clone = cloneState(state)
    cell = getEmptyCell(state)
    validStates = []
    for value in range(1, 10):
        clone[cell[1]][cell[0]] = value
        if isValid(clone):
            validStates.append(cloneState(clone))
    return validStates

def search(state):
    global found, solution
    if found:
        return
    visited.append(state)  # state marked as visited

    # Check for solution
    if getEmptyCell(state) == []:
        solution = state
        found = True
        return
```

```
    for neighbour in getNeighbours(state):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

makeGameGrid(9, 9, 50, Color.red, False, mousePressed = pressEvent)
show()
setTitle("Sudoku")
addStatusBar(30)
visited = []
setBgColor(Color.white)
getBg().setLineWidth(3)
getBg().setPaintColor(Color.red)
for x in range(4):
    getBg().drawLine(150 * x, 0, 150 * x, 450)
for y in range(4):
    getBg().drawLine(0, 150 * y, 450, 150 * y)

startState = [
[0, 6, 0, 7, 9, 8, 0, 1, 2],
[7, 9, 4, 1, 0, 5, 0, 6, 8],
[2, 0, 1, 4, 0, 0, 9, 5, 7],
[0, 0, 0, 2, 1, 0, 5, 0, 0],
[0, 5, 6, 3, 0, 0, 2, 4, 1],
[0, 1, 2, 5, 4, 0, 7, 3, 9],
[6, 3, 0, 8, 7, 4, 0, 0, 0],
[1, 0, 5, 6, 0, 2, 8, 0, 0],
[4, 2, 8, 9, 0, 1, 0, 7, 0]]

fixedLocations = []
for x in range(9):
    for y in range(9):
        if startState[y][x] != 0:
            fixedLocations.append(Location(x, y))

showState(startState)
setStatusText("Press any key to search solution.")
getKeyCodeWait(True)
setStatusText("Searching. Please wait...")
found = False
solution = None
search(startState)
if solution != None:
    showState(solution)
    setStatusText("Solution found")
else:
    setStatusText("No solution")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can also use Sudokus that you find on the Internet or in a puzzle section. If you have enough patience, your program should always be able to find a solution.

You can greatly reduce the computation time if you include the solution strategies that you would also use when solving by hand. It is up to you to improve the algorithm with such heuristic processes.

The process that you use to solve the Sudoku can also be applied to Sudoku puzzles that have non-square cells. For this, you only need to adjust the function *getBlockValues()* accordingly.

THE JEALOUS HUSBANDS

Already in 1613 the mathematician C. G. Bachet, Sieur de Méziriac published a study entitled *Problèmes plaisants et détectables qui se font par les nombres*, in which he describes the puzzle *Les vilains maris jaloux* described by the following in both French and English:

Claude Gaspard Bachet de Méziriac (1581-1638) (© Wiki)

"Trois maris jaloux se trouvent le soir avec leurs femmes au passage d'une rivière, et rencontrent un bateau sans batelier; le bateau est si petit, qu'il ne peut porter plus de deux personnes à la fois. On demande comment ces six personnes passeront de tel sorte qu'aucune femme ne demeure en la compagnie d'un ou de deux hommes, si son mari n'est présent, soit sur l'une des deux rives, soit sur le bateau." (Lit. Édouard Lucas, L'arithmétique amusante" 1885, reprint 2006)

"Three jealous husbands find themselves with their wives at a river crossing in the evening where they find a boat without a boatman. The boat is so small that it only fits two people. This poses the question of how all six people can cross the river without ever putting one of the women in the presence of one or two men without her own husband being present, on either side of the land or in the boat."

You again solve this problem step by step. First you create the user interface (a *GameGrid* works well for this) since the people can be easily represented by sprite images. Add the *GameGrid* actors to an invisible 7x3 grid so that you can draw a river on the middle strip. Since it is only really important if a person is to the left or the right of the river, a binary number is suitable as a data structure where each bit belongs to a particular person. The bit value 0 means that the person is on the left side of the river, and the value 1 means that they are on the right side. Use the bit with the highest value for the boat.

In the simulation program there is a blue, a green, and a red couple that you related to the digits of the binary number as follows:

b6	b5	b4	b3	b2	b1	b0
boat	man_red	female_red	man_green	female_green	man_blue	female_blue

b0 is the bit with the smallest value. If you interpret the state in the decimal system, all numbers between 0 and 127 occur, i.e. there are 128 different states.

It is highly recommended that you take a small detour here so that you can try the coding of the states in a test program. The image jumps from one side of the river to the other when you click on a person or on the boat. The state is written out to the title bar in both its decimal and its binary notation.

You already build a test here with *isStateAllowed(state)* to check if the current situation is legal according to the rules. (You could also first create a prototype without this test.)

```
from gamegrid import *

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in left_locations:
        actor = getOneActorAt(loc)
```

```
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    if loc in right_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    state = 0
    for i in range(7):
        loc = right_locations[i]
        actor = getOneActorAt(loc)
        if actor != None:
            state += 2**(6 - i)
    showState(state)

def stateToString(state):
    return str(bin(state)[2:]).zfill(7)

def showState(state):
    sbin = stateToString(state)
    for i in range(7):
        if sbin[i] == "0":
           actors[i].setLocation(left_locations[i])
        else:
           actors[i].setLocation(right_locations[i])
    setTitle("State: " + str(state) + ", bin: " + stateToString(state))
    if isStateAllowed(state):
        setStatusText("Situation allowed")
    else:
        setStatusText("Situation not allowed")
    refresh()

def isStateAllowed(state):
    print state
    stateStr = stateToString(state)
    mred = stateStr[1] == "1"
    fred = stateStr[2] == "1"
    mgreen = stateStr[3] == "1"
    fgreen = stateStr[4] == "1"
    mblue = stateStr[5] == "1"
    fblue = stateStr[6] == "1"

    if mred and not fred or not mred and fred: # mred/fred not together
        if not fred and (not mgreen or not mblue) or fred and (mgreen or mblue):
            return False
    if mgreen and not fgreen or not mgreen and fgreen:#mgreen/fgreen not together
        if not fgreen and (not mred or not mblue) or fgreen and (mred or mblue):
            return False
    if mblue and not fblue or not mblue and fblue:  # mblue/fblue not together
        if not fblue and (not mred or not mgreen) or fblue and (mred or mgreen):
            return False
    return True

makeGameGrid(7, 3, 50, None, False, mousePressed = pressEvent)
setBgColor(Color.white)
addStatusBar(30)
show()
actors = [Actor("sprites/boat.png"),
   Actor("sprites/man_0.png"), Actor("sprites/woman_0.png"),
   Actor("sprites/man_1.png"), Actor("sprites/woman_1.png"),
   Actor("sprites/man_2.png"), Actor("sprites/woman_2.png")]

left_locations =  [Location(2, 0),
                   Location(2, 1), Location(2, 2),
                   Location(1, 1), Location(1, 2),
                   Location(0, 1), Location(0, 2)]
```

```
right_locations = [Location(4, 0),
                   Location(4, 1), Location(4, 2),
                   Location(5, 1), Location(5, 2),
                   Location(6, 1), Location(6, 2)]

for i in range(7):
    addActorNoRefresh(actors[i], left_locations[i])
for i in range(3):
   getBg().fillCell(Location(3, i), Color.blue)
refresh()

startState = 0
showState(startState)
```

In the next step you implement the backtracking algorithm, as you know it from **chapter 10.3.** Again, you first have to determine all possible successive states for a given state in *getNeighbours(state)*. You begin as follows: First you distinguish whether the boat is located on the left side of the river (*state < 64*) or the right (*state >=64*). Then you determine all people who are available to move across the river either as an individual or in a two person team from the lists *li_one* and *li_two.* When using *removeForbiddenTransfers()* you also must consider that a woman can never be in the boat with a man that is not her husband.

You implement the backtracking in its familiar form in *search().* You copy the solutions into a list named *solutions*, so that you can access them at the end of the search process to examine the solutions. Thereby you first write out the number of found solutions and then select the shortest one which can then be run through using key press.

```
from gamegrid import *
import itertools

def pressEvent(e):
    global state
    loc = toLocationInGrid(e.getX(), e.getY())
    if loc in left_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    if loc in right_locations:
        actor = getOneActorAt(loc)
        if actor != None:
            x = 6 - actor.getX()
            y = actor.getY()
            actor.setLocation(Location(x, y))
    state = 0
    for i in range(7):
        loc = right_locations[i]
        actor = getOneActorAt(loc)
        if actor != None:
            state += 2**(6 - i)
    setTitle("State: " + str(state) + ", bin: " + stateToString(state))
    if isStateAllowed(state):
        setStatusText("Situation allowed")
    else:
        setStatusText("Situation not allowed")
    refresh()

def stateToString(state):
    return str(bin(state)[2:]).zfill(7)

def showState(state):
    sbin = stateToString(state)
    for i in range(7):
        if sbin[i] == "0":
           actors[i].setLocation(left_locations[i])
```

```
        else:
            actors[i].setLocation(right_locations[i])
    refresh()

def getTransferInfo(state1, state2):
    state1 = state1 & 63
    state2 = state2 & 63
    mod = state1 ^ state2
    passList = []
    for n in range(6):
        if mod % 2 == 1:
            if n // 2 == 0:
                couple = "blue"
            elif n // 2 == 1:
                couple = "green"
            elif n // 2 == 2:
                couple = "red"
            if n % 2 == 0:
                passList.append("f" + couple)
            else:
                passList.append("m" + couple)
        mod = mod // 2
    return passList

def getTransferSequence(solution):
    transferSequence = []
    oldState = solution[0]
    for state in solution[1:]:
        transferSequence.append(getTransferInfo(oldState, state))
        oldState = state
    return transferSequence

def isStateAllowed(state):
    stateStr = stateToString(state)
    mred = stateStr[1] == "1"
    fred = stateStr[2] == "1"
    mgreen = stateStr[3] == "1"
    fgreen = stateStr[4] == "1"
    mblue = stateStr[5] == "1"
    fblue = stateStr[6] == "1"

    if mred and not fred or not mred and fred:  # mred/fred not together
        if not fred and (not mgreen or not mblue) or fred and (mgreen or mblue):
            return False
    if mgreen and not fgreen or not mgreen and fgreen:#mgreen/fgreen not together
        if not fgreen and (not mred or not mblue) or fgreen and (mred or mblue):
            return False
    if mblue and not fblue or not mblue and fblue:  # mblue/fblue not together
        if not fblue and (not mred or not mgreen) or fblue and (mred or mgreen):
            return False
    return True

def removeForbiddenTransfers(li):
    forbiddenPairs = [(0, 3), (0, 5), (1, 2), (2, 5), (1, 4), (3, 4)]
    allowedPairs = []
    for pair in li:
        if pair not in forbiddenPairs:
            allowedPairs.append(pair)
    return allowedPairs

def getNeighbours(state):
    neighbours = []
    li_one = []  # one person in boat
    bin = stateToString(state)
    if state < 64:  # boat at left
        for i in range(6):
            if bin[6 - i] == "0":
                li_one.append(i)
        li_two = list(itertools.combinations(li_one, 2)) #two persons in boat
```

```
        li_two = removeForbiddenTransfers(li_two)
    else: # boat at right
        for i in range(6):
            if bin[6 - i] == "1":
                li_one.append(i)
        li_two = list(itertools.combinations(li_one, 2))
        li_two = removeForbiddenTransfers(li_two)

    li_values = []
    if state < 64:  # boat at left, restrict to two persons transfer
        for li in li_two:
            li_values.append(2**li[0] + 2**li[1] + 64)
    else:  # boat at right, one or two persons transfer
        for i in li_one:
            li_values.append(2**i + 64)
        for li in li_two:
            li_values.append(2**li[0] + 2**li[1] + 64)

    for value in li_values:
        v = state ^ value
        if isStateAllowed(v):  # restrict to allowed states
            neighbours.append(v)
    return neighbours

def search(state):
    visited.append(state)  # state marked as visited

    # Check for solution
    if state == targetState:
        solutions.append(visited[:])

    for neighbour in getNeighbours(state):
        if neighbour not in visited: # Check if already visited
            search(neighbour) # recursive call
    visited.pop()

nbSolution = 0
makeGameGrid(7, 3, 50, None, False, mousePressed = pressEvent)
addStatusBar(30)
setBgColor(Color.white)
setTitle("Searching...")
show()
visited = []
actors = [Actor("sprites/boat.png"),
   Actor("sprites/man_0.png"), Actor("sprites/woman_0.png"),
   Actor("sprites/man_1.png"), Actor("sprites/woman_1.png"),
   Actor("sprites/man_2.png"), Actor("sprites/woman_2.png")]

left_locations =  [Location(2, 0),
                   Location(2, 1), Location(2, 2),
                   Location(1, 1), Location(1, 2),
                   Location(0, 1), Location(0, 2)]
right_locations = [Location(4, 0),
                   Location(4, 1), Location(4, 2),
                   Location(5, 1), Location(5, 2),
                   Location(6, 1), Location(6, 2)]

for i in range(7):
    addActorNoRefresh(actors[i], left_locations[i])
for i in range(3):
   getBg().fillCell(Location(3, i), Color.blue)
refresh()

startState = 0
targetState = 127
solutions = []
search(startState)

maxLength = 0
```

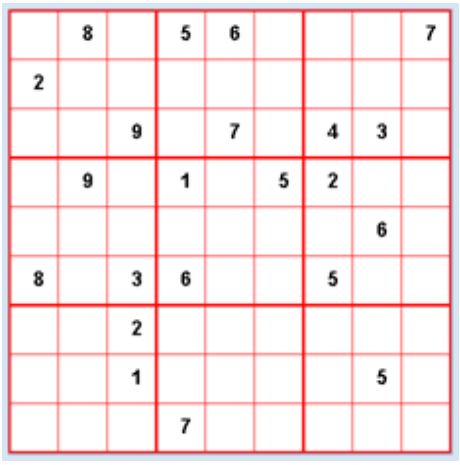
```
maxSolution = None
minLength = 100
minSolution = None
for solution in solutions:
    if len(solution) > maxLength:
        maxLength = len(solution)
        maxSolution = solution
    if len(solution) < minLength:
        minLength = len(solution)
        minSolution = solution
setStatusText("#Solutions: " + str(len(solutions)) + ", Min Length: "
            + str(minLength) + ", Max Length: " + str(maxLength))

setTitle("Press key to cycle")
oldState = startState
for state in minSolution[1:]:
    getKeyCodeWait(True)
    showState(state)
    info = getTransferInfo(oldState, state)
    setTitle("#Transferred: " + str(info))
    oldState = state
setTitle("Done. #Boat Transfers: " + str((len(minSolution) - 1)))
```

MEMO

Puzzles of this type have the feature that you do not know whether they have exactly one or multiple solutions from the start. It turns out that it is much easier for us humans to find a solution, than it is to prove that there is no solution. For the computer which systematically searches for all solutions with an *exhaustive search,* the search for all solutions is generally not a problem except that it can take a long time. This is unfortunately already the case with relatively simple games, due to combinatorial explosion, so we again bump into the limits of using the computer. It is interesting that Bachet had already published the solution with minimum amount of 11 crossings, which is also what your computer program finds. He argued so skillfully crossing by crossing that the strategy of solving the puzzle appears to be evident. However, whether he found the solution first by "trial and error", and only later wrote the arguments for it, remains undecided.

EXERCISES

1. Find a solution for the following *X-Sudoku*, in which the 9 numbers can also only appear once on either of the two main diagonals.



	8		5	6				7
2								
		9		7		4	3	
	9		1		5	2		
							6	
8		3	6			5		
		2						
		1					5	
			7					

2. Show that there is no solution to the problem of "The Jealous Husbands" if only a single person is allowed to cross the river from right to left at a time.

11.2 PITFALLS, RULES & TRICKS

INTRODUCTION

As in any programming language, there are also some pitfalls in *Python* that even experienced programmers have to deal with sometimes. You can deal with them too if you know them as a potential source of danger.

THE VARIABLE MEMORY MODEL OF PYTHON

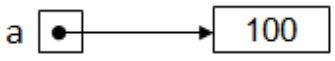
In **chapter 2.6**the concept of variables was introduced by a visual representation, the so-called box metaphor. A variable declaration, for instance a = 100, was considered to be a reservation of space in computer memory to hold the number 100, simular to creating a drawer or a box and putting the number 100 inside. Similar to mathematics, the letter *a* stands for a variable identifier (name or place holder). However this simple idea is not entirely correct in Python, since all data, including numbers, are regarded as objects and an object does not only have a value, but also define "behavior" using *functions* (or *methods*). As an example number objects "knows" how to add values.
The following lines demonstrate this concept:

```
>>> a = 100
 < a: 100
>>> a.__add__
 < built-in method __add__ of int object at 0x2>
```

Sometimes we say that 2 is the *address* of the object, in Python called *id* (identifier).

```
>>> id(a)
 < 2
```

In Python, by assigning *a* value to a variable, you create a name *a* that *points to* (or *refers*) to an *int* object, that is stored at memory address 0x2. Therefore *a* is also called an "*alias*" (or on other programming languages a *reference* or *pointer*). This situation can be represented symbolically:



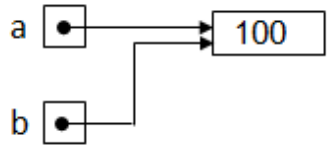
The difference to the box metaphor becomes clearly visible in the following assignment:

```
>>> b = a
 < b: 100
```

Actually this statement does not create a second box and put the number 100 inside, but simply creates a **second** alias *b* that refers to the **same** object that holds 100, as shown here:

```
>>> is(b)
 < 2
```

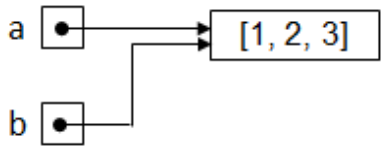
The situation looks like this:



The awareness of this concept becomes most important when the data is not a simply number, but a structured data type (like a list) as shown in the following example.
First you define a list *a* with a simple content. Then you create a second list *b* by assigning *b* to a:

```
>>> a = [1, 2, 3]
 < a: [1, 2, 3]
>>> b = a
 < b: [1, 2, 3]
```

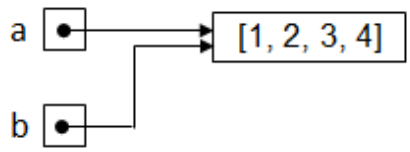
As you suspect correctly the situation can be represented as follows



If you modify now the list using the alias *b*

```
>>> b.append(4)
>>> b
 < [1, 2, 3, 4]
```

it's like



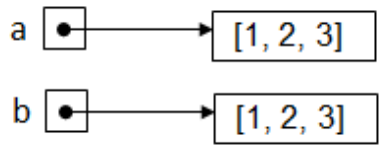
Therefore you also changed the list referenced by *a* (it is the same)!

```
>>> a
 < a: [1, 2, 3, 4]
```

This mutual dependency of the two variables *a* and *b* causes many subtle programming errors. But there is no danger if you define a completely new variable *b* because now you create a new object and the two objects *a* and *b* are completely independent.

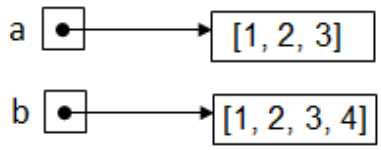
```
>>> a = [1, 2, 3]
 < a: [1, 2, 3]
>>> b = a
>>> b = [1, 2, 3]
 < b: [1, 2, 3]
```

or symbolically:



Now if you modify the list *b*:

```
>>> b.append(4)
```



the list a remains unchanged as you can easily check:

```
>>> b
 < b: [1, 2, 3, 4]
>>> a
 < a: [1, 2, 3]
```

For simple data types like numbers, this mutual dependency does not harm because if you modify a number, a new number is automatically created. Indeed after the new assignment of *b*, a new object is created in memory. The alias that referred to value 100 after the second line *b = a* now refers to a new object 200 after the third line.

```
>>> a = 100
 < a: 100
>>> b = a
 < b: 100
>>> b = 200
 < b: 200
>>> a
 < 100
>>>
```

The following example demonstrates once again that the assignment of list variable is dangerous since the **content of the list is not copied** into the new list.

```
>>> myGarden = ["Rose", "Lotus"]
>>> yourGarden = myGarden
>>> yourGarden[0] = "Hibiskus"
>>> myGarden
 < ["Hibiskus", "Lotus"]
>>> yourGarden
 < ["Hibiskus", "Lotus"]
```

Rule 1a:
The copy operation using the equal sign is unproblematic for numbers, strings, bytes, and tuples. For other data types, it is usually wrong.

If you want to create an independent copy (also called a ***clone***) for mutable data types, for example of lists, you have to either copy the elements explicitly into a new variable with your own code, or you can use the function *deepcopy()* from the module *copy*:

```
>>> import copy
>>> myGarden = ["Rose", "Lotus"]
>>> yourGarden = copy.deepcopy(myGarden)
>>> yourGarden[0] = "Hibiskus"
>>> myGarden
 < ["Rose", "Lotus"]
>>> yourGarden
 < ["Hibiskus", "Lotus"]
```

As a consequence we formulate the following rule:

Rule 1b:
The copy operation using the equal sign is is usually wrong, except when the data is immutual (see below).

Rule 1 is often ignored in the context of parameter passing. Each time you pass a function a mutable data type, the function can change the values without any problems.

```
>>> def show(garden)
>>>     print "garden:", garden
>>>     garden[0] = "Hibiskus"
>>> myGarden = ["Rose", "Lotus"]
>>> show(myGarden)
>>> myGarden
 < ["Hibiskus", "Lotus"]
```

After calling *show(),* the values of the passed parameters changed! Just like in medicine, it is usually an unexpected and unwanted adverse effect or **side effect**.

Rule 2:
In good programming style, you should not change the parameters passed into a function to avoid side effects.

PACKING & UMPACKING

At first glance, tuples do not seem to differ significantly from lists. As a matter of fact, tuples are basically immutable lists and thus all list operations that do not change the list, are also applicable to tuples. However, there are special notations techniques with tuples using commas.

Round parentheses can be omitted in the generation of tuples:

```
>>>   t = 1, 2, 3
>>>   t
>>>   (1, 2, 3)
```

In this case, the comma is used as a syntax character to separate the elements from each other. This is called **automatic packing**. You can smartly make use of this to return multiple function values as tuples:

```
>>> import math
>>> def sqrt(x):
>>>     y = math.sqrt(x)
>>>     return y, -y
>>> sqrt(4)
 < (2,0, -2,0)
```

You can also use the comma operator on the left side when defining variables in case a tuple is placed after the equal sign. In this case, one speaks of **automatic unpacking**:

```
>>> import math
>>> def sqrt(x):
>>>     y = math.sqrt(x)
>>>     return y, -y
>>> y1, y2 = sqrt(2)
>>> y1
 < 1.41421356237330951
>>> y2
 < -1.41421356237330951
```

Packing is convenient for defining multiple variables simultaneously:

```
>>> a, b, c = 1, 2, 3
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

In the following, the three numbers are packed on the right side and unpacked again on the left side:

```
>>> t = 1, 2, 3
>>> a, b, c = t
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

By the way, unpacking also works with lists:

```
>>> li = [1, 2, 3]
>>> a, b, c = li
>>> a
 < 1
>>> b
 < 2
>>> c
 < 3
```

Using this syntax, two numbers can be permuted in an elegant way without requiring an auxiliary variable:

```
>>> li = [1, 2, 3]
>>> a, b = b, a
>>> a
 < 2
>>> b
 < 1
```

MUTUAL AND IMMUTUAL DATA TYPES

To improve data security by undesirable side effects, Python relies on the concept of alterable (mutable) and non-alterable (immutable) data types. The latter include the numbers, strings (str), and byte tuple. For example changing a letter in a string causes an error message:

```
>>> s = "abject"
 < s: "abject"
>>> s[0] = "o"
 < TypeError:can't assign to immutable object
```

To modify s, you need to redefine the whole string:

```
>>> s = "abject"
 < s: "abject"
>>> s = "object"
 < s: "object"
```

Now a new string object is created and the reference to the old is lost (the memory space is freed by an internal garbage colllector).

TWO-DIMENSIONAL LISTS, MATRICES

Matrices are constructed as arrays in many programming languages. The rows of the matrix are arrays and the matrix itself is an array of these row arrays. It is straightforward to use lists instead of arrays in *Python*. However, you have to pay close attention since lists do not behave like elementary data types (immutable), but rather like reference types. You already get into a known pitfall during the creation of the matrix. Without knowing it, you generate a 3x3 matrix with zeros in the *Python* console.

```
>>> A = [[0] * 3] * 3
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

You now change the last value of the first row using an allocation. You are probably surprised to notice that all other rows change too.

```
>>> A[0][2] = 1
>>> A
 < [[0, 0, 1],
    [0, 0, 1],
    [0, 0, 1]]
```

What happened there? Thinking about it a little will start you off. Generating *A* could have also been done in two steps:

```
>>> z = [0] * 3
>>> A = [z] * 3
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
```

First, a list *z* is generated with three zeros. Then a list *A* is made with three times the same line reference. All refer to the same list! If you change one of them, the others will also be affected.

Rule 3:
Never use the list multiplication sign for nested lists.

To get around the pitfalls, you can use list comprehension. As you can test out, the matrix now behaves properly:

```
>>> A = [[0 for x in range(3)] for y in range(3)]
>>> A
 < [[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]]
>>> A[0][2] = 1
>>> A
 < [[0, 0, 1],
    [0, 0, 0],
    [0, 0, 0]]
```

11.3 BUGS & DEBUGGING

INTRODUCTION

There is no absolute perfection. You can assume that practically every software has its faults. Instead of calling these *mistakes*, in programming we call them *bugs*. These manifest themselves when, for example, the program produces incorrect results under certain circumstances or when it even crashes. Troubleshooting, called debugging, is therefore almost as important as writing code. It is understood, of course, that anyone who writes code should do their best to avoid bugs. Knowing that bugs are everywhere, you should be careful and program defensively. If you are not sure if an algorithm or a piece of the code is correct, it is better if you engage with it especially intensely instead of rushing through it as quickly as possible or putting it off for later. Nowadays there is a lot of software whose main function is dealing with large sums of money or even human lives that are at stake. As a programmer of such mission critical software, you need to be absolutely sure and take the responsibility that every line of code works correctly. Quick hacks and the principle of trial and error are not appropriate there.

The crucial objective of developing algorithms and large software systems is to produce programs that are as error-free as possible. There are many approaches for this: You could try to prove the correctness of programs in a mathematically precise manner without using the computer, although this can only be done with short programs. Another possibility that exists is to limit the syntax of the programming language so that the programmer can definitely not make certain mistakes. The most important example is the elimination of pointer variables (pointers) that may refer to undefined or wrong objects when not used properly. This is why there are programming languages with many restrictions of this kind, and others that give the programmer a considerable amount of freedom and which are therefore less secure. *Python* belongs to the class of more liberal programming languages and is based on the motto:

"We are all adults and decide for ourselves what we do and what we shouldn't."
or *"After all, we are all consenting adults here".*

An important principle for creating software as error-free as possible is Design by Contract (DbC). It goes back to Bertrand Meyer at ETH Zürich, the father of the programming language Eiffel. Meyer views software as an *agreement* between the programmer A and the user B, where B could again be a programmer who uses the modules and libraries designed by A. In a *contract,* A determines under which conditions (preconditions) its modules will provide the correct results and then describes it exactly (postconditions). In other words: User B complies with the preconditions so that they have the guarantee from A that they will receive a result in accordance with the postconditions. B does not need to know the implementation of the modules and A can change this at any time without affecting B.

ASSERTIONS

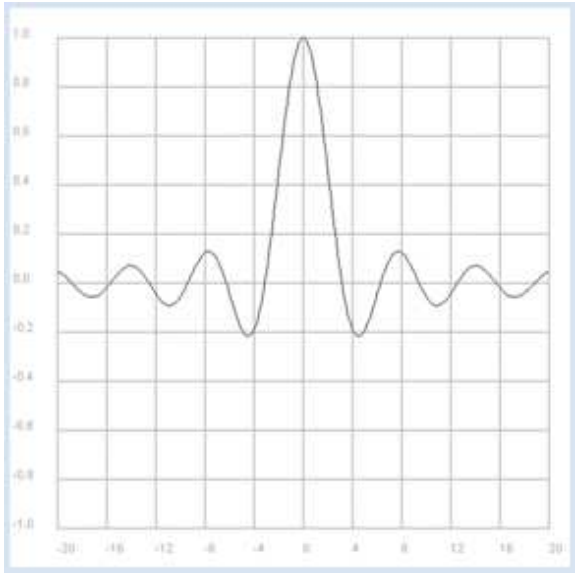
Since A, the producer of the module, certainly does not really trust the user B, A incorporates tests into their software, which check the required preconditions. These are called **assertions**. If the assertions are not observed, the program usually terminates with an error message. (It is also conceivable that the error is only caught without the program terminating.) In this case, the following principle of software development comes into play:

Forcing a program termination with a clear description of the possible cause (an error message) is better than an incorrect result.

As programmer A, you write a function *sinc(x)* (sinus cardinalis) in your example, which plays an important role in signal processing. It reads as follows:

$$f(x) = \frac{\sin x}{x}$$

As user B, you want to graphically represent the function values in the range from x = -20 to x = 20.



```
from gpanel import *
from math import pi, sin

def sinc(x):
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 0.1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

At first glance there do not seem to be any problems, but then if you (as the user B) change the increment of x to 1, there is a bad crash.

```
def sinc(x):
    y = sin(x) / x
```

Division durch Null ist nicht möglich!

A way to solve the problem is for A to require the precondition that x is not 0, and output an assertion that describes the error.

```
from gpanel import *
from math import pi, sin

def sinc(x):
    if x == 0:
        return 1.0
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
```

```
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

The error message is now much better, since it says exactly where the error occurs.

It would be even better if A wrote the function so that the limit value 1 of the function is returned when x = 0.

```
from gpanel import *
from math import pi, sin

def sinc(x):
    if x == 0:
        return 1.0
    y = sin(x) / x
    return y

makeGPanel(-24, 24, -1.2, 1.2)
drawGrid(-20, 20, -1.0, 1, "darkgray")
title("Sinus Cardinalis: y = sin(x) / x")

x = -20
dx = 1
while x <= 20:
    y = sinc(x)
    if x == -20:
        move(x, y)
    else:
        draw(x, y)
    x += dx
```

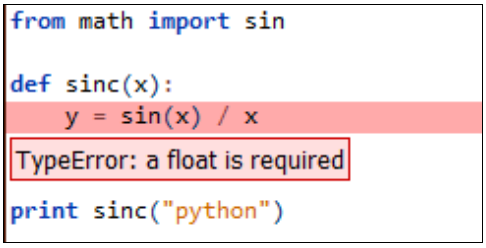
Highlight program code (Ctrl+C copy, Ctrl+V paste)

This code, however, contradicts the basic rule that equality tests with floats are dangerous due to possible rounding errors. Even better would be to test it with an epsilon boundary:

```
def sinc(x):
    epsilon = 1e-100
    if abs(x) < epsilon:
        return 1.0
```

Your function *sinc(x)* is still not secure, however, since another precondition should be that x is a number. If you, for instance, call *sinc(x)* with the value *x = "python"*, it results again in a nasty runtime error.

```
from math import sin

def sinc(x):
    y = sin(x) / x
    return y

print sinc("python")
```



This shows the advantage of programming languages with variable declarations, as this error would be already discovered as a syntax error before the execution of the program.

WRITING OUT DEBUGGING INFORMATION

Successful programmers are known for their ability to eliminate errors quickly [**more...**]. Since we all learn from our mistakes, consider the following program that should exchange the values of two variables. There is a bug, because it outputs 2,2.

```
def exchange(x, y):
    y = x
    x = y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

A well-known and simple strategy for finding bugs is to write out the current values of certain variables to the console:

```
def exchange(x, y):
    print "exchange() with params", x, y
    y = x
    x = y
    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Hence, it becomes obvious where the error occurred and how you can easily fix it.

```
def exchange(x, y):
    print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
    print "exchange() returning", x, y
    return x, y
```

```
a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

Now that the error is fixed, the additional debug lines in the code are unnecessary. Instead of deleting them, you can simply comment them out using the # symbol since you might need them again later.

```
def exchange(x, y):
#    print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
#    print "exchange() returning", x, y
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

It is smart to use debug flags with which you can activate or deactivate debugging information in different places.

```
def exchange(x, y):
    if debug: print "exchange() with params", x, y
    temp = y
    y = x
    x = temp
    if debug: print "exchange() returning", x, y
    return x, y

debug = False
a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

In *Python*, as you probably already know, you can swap variable values in an elegant way without using an auxiliary variable. For this, you can use the automatic packing/unpacking of tuples.

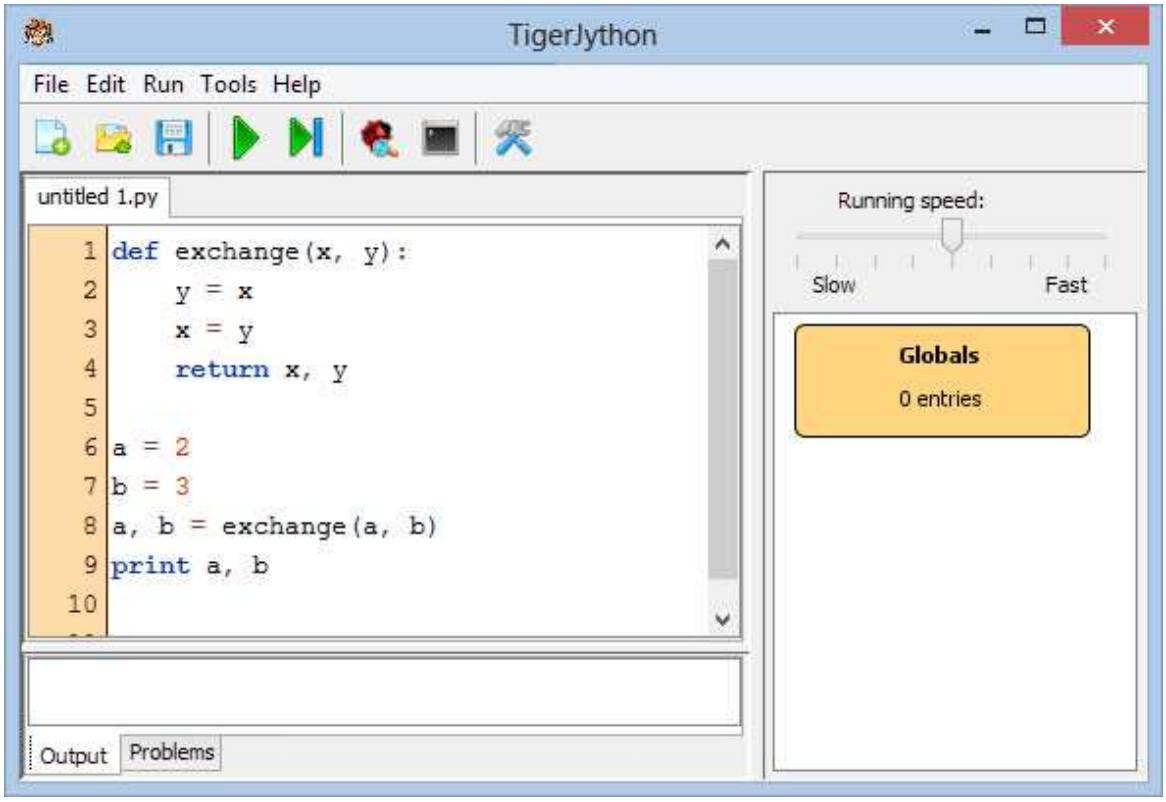
```
def exchange(x, y):
    x, y = y, x
    return x, y

a = 2
b = 3
a, b = exchange(a, b)
print a, b
```

USING THE DEBUGGER

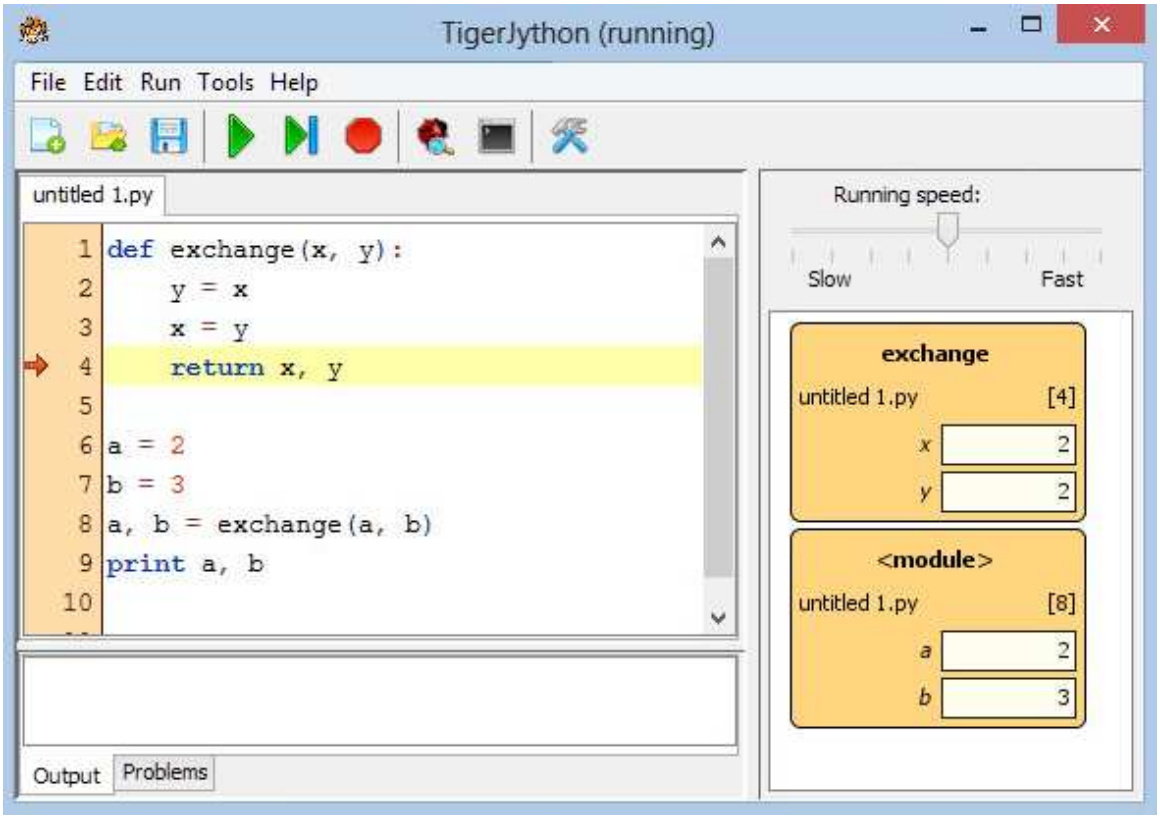
Debuggers are important tools for developing large program systems. You can run a program slowly with them, and even run it step by step. So to speak, the enormous execution speed of the computer is adjusted to the limited human cognitive ability. A confortable debugger is ncluded in *TigerJython* which can help you to better understand the program sequence in a correct program. You are now going to analyze the above defective program with the debugger.

After you have taken it to the editor, click on the debugger button.

You can run the program step by step with the single step button and observe the variables in the debugger window. After clicking 8 times you see that both variables x and y have the value 2 before returning from the function *exchange()*..

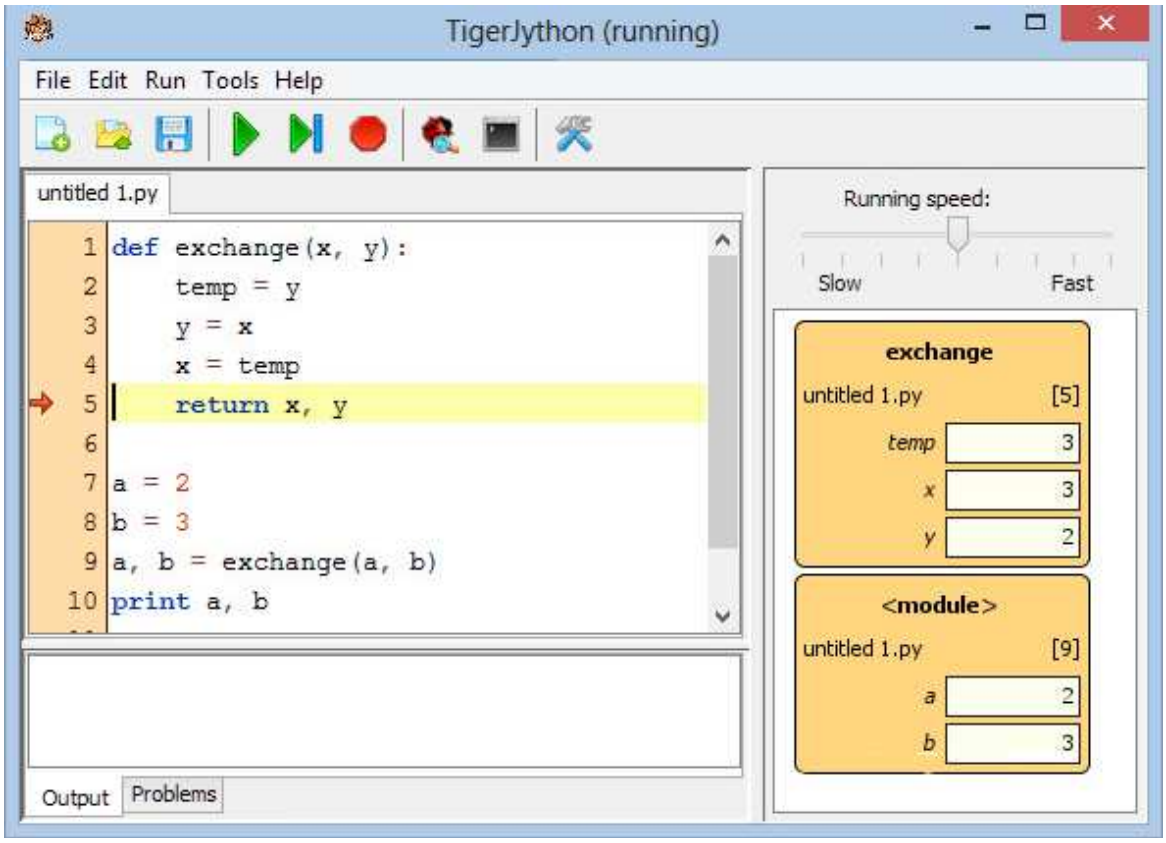


If you now run the program with the fixed bug, you can observe how the variables are assigned to one another in *exchange()*, which finally leads to the correct results. The short life span of the local variables x and y are clearly visible, as opposed to the global variables a and b.



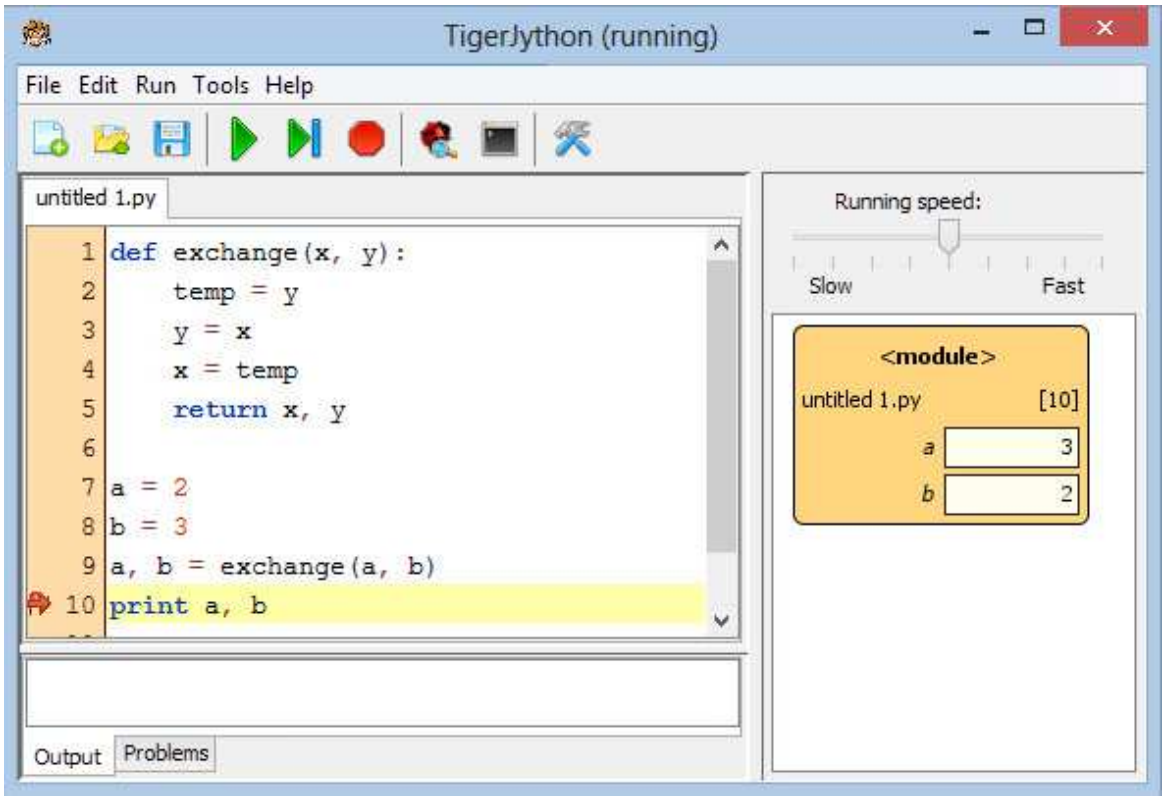
You can also set breakpoints in the program so that you do not have to run through tedious non-critical parts of the program in the single step mode. To do this, click on the far left of the line number column. A flag icon appears which marks the breakpoint. When you press the Run

button, the program runs up to this point and then stops.



This gives you the chance to inspect the current state of the variables. By clicking on the Run button again you can continue to run through the program, or you can investigate it gradually with the single step button.

You can also set multiple breakpoints. In order to delete a breakpoint, simply click on the flag icon.



CATCHING ERRORS WITH EXCEPTIONS

The method of using exceptions to catch errors is classic. To do this, you put the critical program code in a try block. If the error occurs, the block is abandoned and the program continues to run in the except block, where you can react to the error in an appropriate way. In the worst case, you can stop the program execution by calling *sys.exit()*.

You can, for example, catch the error if the parameter in *sinc(x)* is not of a numeric data type.

```
from sys import exit
from math import sin

def sinc(x):
    try:
        if x == 0:
            return 1.0
        y = sin(x) / x
    except TypeError:
        print "Error in sinc(x). x =", x, "is not a number"
        exit()
    return y

print sinc("python")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

The postcondition for *sinc(x)* could also mean that the returned value is *None* if the parameter has an incorrect type. It is then up to the user to deal with this error accordingly.

```
from math import sin

def sinc(x):
    try:
        if x == 0:
            return 1.0
        y = sin(x) / x
    except TypeError:
        return None
    return y

y = sinc("python")
if y == None:
   print "Illegal call"
else:
   print y
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

11.4 PARALLEL PROCESSING

INTRODUCTION

According to the von Neumann model, one can imagine a computer as a sequential machine that, based on a program, executes statement by statement in time intervals. In this model there are no simultaneous actions, so no parallel processing and no concurrency. In daily life however, parallel processes are omnipresent: Every living being exists as an independent individual and many processes run simultaneously in the human body.

The advantages of parallel processing are evident: It brings on a huge performance boost, since tasks are solved in equal time slices. Moreover, the redundancy and the odds of survival increase because the failure of a particular component does not automatically lead to the failure of the entire system.

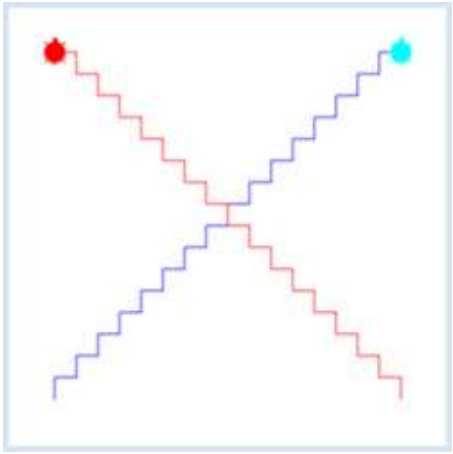
However, parallelizing algorithms is a challenging task, which despite great efforts is still in its infancy. The problem is mainly the fact that the sub-processes usually use shared resources and have to wait on the results of other processes.

A *thread* is code running in parallel within the same program and a *process* is code that is executed in parallel by the operation system. *Python* provides a good support of both types of parallelism. Here, however, we only consider the use of multiple threads, so **multithreading**.

MULTITHREADING IS EASIER THAN IT SEEMS

In *Python*, it is very easy to run the code of one of your functions from its own thread: To do this you import the module *threading* and pass *start_new_thread()* the function name as well as possible parameter values that you pack into a tuple. The thread begins running immediately and executes the code of your function.

In your first program with threads, two turtles should draw a staircase independently of each other. To do this, you write an arbitrarily named function, in this case denoted by *paint()*, which may also have parameters, such as here the turtle and a flag that indicates whether the turtle draws a staircase to the left or the right. You then *pass* the function name and a tuple with the parameter values (the turtle and the flag) to the function *thread.start_new_thread()*. And now off you go!



```
from gturtle import *
import thread

def paint(t, isLeft):
    for i in range(16):
        t.forward(20)
        if isLeft:
            t.left(90)
        else:
            t.right(90)
```

```
        t.forward(20)
        if isLeft:
            t.right(90)
        else:
            t.left(90)

tf = TurtleFrame()
john = Turtle(tf)
john.setPos(-160, -160)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(160, -160)
thread.start_new_thread(paint, (john, False))
thread.start_new_thread(paint, (laura, True))
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

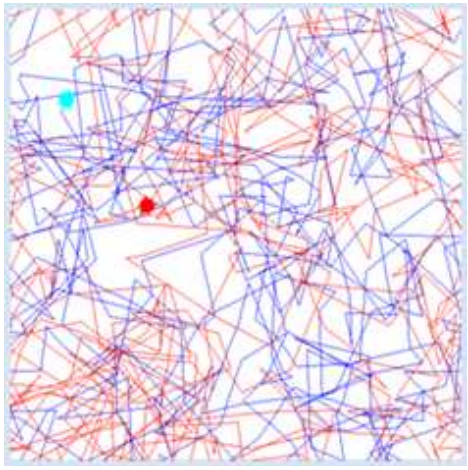
MEMO

In order to move both turtles in the same window you use a *TurtleFrame tf* and pass it to the turtle constructor.

A new thread is created and immediately started with *start_new_thread()*. The thread is terminated as soon as the passed function returns.

The parameter list has to be specified as a tuple. Be aware that an empty tuple () must be passed for a function without parameters, and that a tuple with a single element *x* should be written as *(x,) instead of (x)*

CREATING AND STARTING THREADS AS A CLASS INSTANCE



You get a little more leeway when you define a separate class that is derived from the class *Thread*. In this class, you overwrite the method *run()* which contains the code to be executed.

To start the thread, you first create an instance and call the method *start()*. The system will then execute the method *run()* automatically in a new thread and stop the thread once *run()* is done.

```
from threading import Thread
import random
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
            self.t.forward(150 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
```

```
john = Turtle(tf)
john.wrap()
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.wrap()
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
thread1.start()
thread2.start()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

Even with a multiprocessor system the code is not really run in parallel, but rather in successive time slices. Therefore, it usually merely consists of quasi-parallel data processing. However, it is important that the allocation of the processor to the threads takes place at unpredictable points in time, so somewhere in the middle of your code. The breakpoint and the local variables are automatically rescued when your thread is interrupted, and restored when it continues, but problems may arise if in the meantime other threads change shared global data. This also applies to the content of a graphics window. Therefore, it is not self-evident that the two turtles do not get in the way of each other [**more...**].

You can see, the main part of the program finishes running, but the two threads keep performing their tasks until the window is closed.

ENDING THREADS

Once initiated, a thread can not be stopped directly using a method from outside, such as by another thread. In order to stop a thread, it must be ensured that the method *run()* comes to the end. That is why a non-breakable while loop in the method *run()* of a thread is never a good idea. Instead, you should use a global boolean variable *isRunning* for the while loop that is normally set to True, but that can also be set to False from another thread.

Both turtles execute a random movement in your program until one of the two moves out of a circular area.

```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while isRunning:
            self.t.forward(50 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
```

```
thread2 = TurtleAnimator(laura)
isRunning = True
thread1.start()
thread2.start()

while isRunning and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isRunning = False
    time.sleep(0.001)
tf.setTitle("Limit exceeded")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

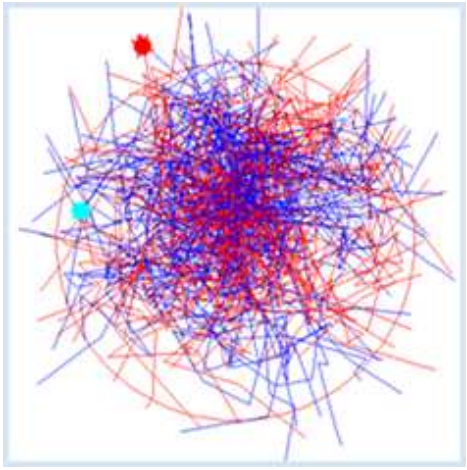
MEMO

You should never use a "tight" loop that does not perform actions in the body, because you are wasting a lot of processor time. Always set a small waiting time of at least a few milliseconds using *time.sleep()*, *Turtle.sleep*(), or *GPanel.delay()*.

Once a thread has ended it can not be initiated again. If you try to call *start()* once again, there is an error message.

STOPPING AND CONTINUING THREADS

In order to stop a thread only for a certain time, you can skip the actions in *run()* using a global flag *isPaused* and continue again later using *isPaused* = *False*.



```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
            if isPaused:
                Turtle.sleep(10)
            else:
                self.t.forward(100 * random.random())
                self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
```

```
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isPaused = False
thread1.start()
thread2.start()

tf.setTitle("Running")
while not isPaused and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isPaused = True
        tf.setTitle("Paused")
        Turtle.sleep(2000)
        laura.home()
        john.home()
        isPaused = False
        tf.setTitle("Running")
    time.sleep(0.001)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

It is even smarter to stop the thread using *Monitor.putSleep()* and then continue later using *Monitor.wakeUp()*.

```
from threading import Thread
import random, time
from gturtle import *

class TurtleAnimator(Thread):
    def __init__(self, turtle):
        Thread.__init__(self)
        self.t = turtle

    def run(self):
        while True:
            if isPaused:
                Monitor.putSleep()
            self.t.forward(100 * random.random())
            self.t.left(-180 + 360 * random.random())

tf = TurtleFrame()
john = Turtle(tf)
laura = Turtle(tf)
laura.setColor("red")
laura.setPenColor("red")
laura.setPos(-200, 0)
laura.rightCircle(200)
laura.setPos(0, 0)
thread1 = TurtleAnimator(john)
thread2 = TurtleAnimator(laura)
isPaused = False
thread1.start()
thread2.start()

tf.setTitle("Running")
while not isPaused and not tf.isDisposed():
    if laura.distance(0, 0) > 200 or john.distance(0, 0) > 200:
        isPaused = True
        tf.setTitle("Paused")
        Turtle.sleep(2000)
        laura.home()
        john.home()
        isPaused = False
        Monitor.wakeUp()
        tf.setTitle("Running")
    time.sleep(0.001)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

A thread can stop itself using the blocking method *Monitor.putSleep()* so that it does not waste any computing time. Another thread can activate it again using *Monitor.wakeUp()*, i.e. the blocking method *Monitor.putSleep()* returns.

WAITING ON THREAD RESULTS

In this program you employ a worker to calculate the sum of natural numbers from 1 to 1,000,000 simply by adding. You wait in the main program until the job is done and then determine the time that was required. Since this can vary slightly, you let the work be performed 10 times by a worker thread. In order to wait for the end of the thread, you use *join()*.

```
from threading import Thread
import time

class WorkerThread(Thread):
     def __init__(self, begin, end):
         Thread.__init__(self)
         self.begin = begin
         self.end = end
         self.total = 0

     def run(self):
         for i in range(self.begin, self.end):
             self.total += i

startTime = time.clock()
repeat 10:
    thread = WorkerThread(0, 1000000)
    thread.start()
    thread.join()
    print thread.total
print "Time elapsed:", time.clock() - startTime, "s"
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

Just like in real life, you can distribute tedious work to multiple workers. If you use two worker threads for this, so that each of them performs half of the work, you will need to wait for both to finish before you add up the sum.

```
from threading import Thread
import time

class WorkerThread(Thread):
     def __init__(self, begin, end):
         Thread.__init__(self)
         self.begin = begin
         self.end = end
         self.total = 0

     def run(self):
         for i in range(self.begin, self.end):
             self.total += i

startTime = time.clock()
repeat 10:
    thread1 = WorkerThread(0, 500000)
    thread2 = WorkerThread(500000, 1000000)
    thread1.start()
    thread2.start()
```

```
    thread1.join()
    thread2.join()
    result = thread1.total + thread2.total
    print result
print "Time elapsed:", time.clock() - startTime, "s"
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You could also set a global flag *isFinished() to True* when the threads terminate, and test this flag in a waiting loop in the main part of the program. However, this solution is less elegant than just using *join()* because you are wasting computing time by constantly testing the flag.

The computing time is somewhat smaller when using two threads. The difference is small since the program does not really run them in parallel, but rather in sequential time slices and since a certain time is required for switching threads **more...**]

CRITICAL SECTIONS AND LOCKS

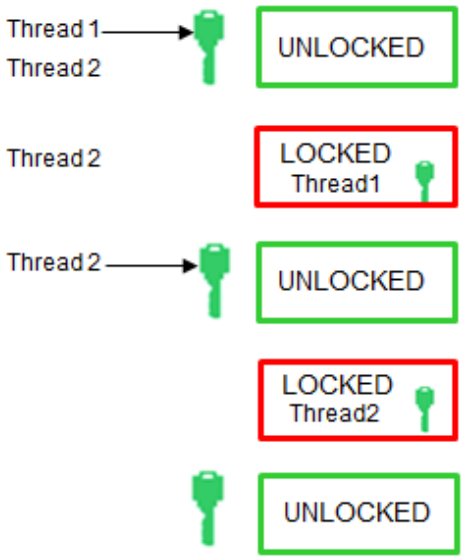
Because threads execute code virtually independently, it is awkward if multiple threads modify shared data. To avoid collisions between threads, actions belonging together are encapsulated in a so-called critical program block and provided with a protection that ensures that the block is only executed uninterruptedly as a whole (*atomically*). If another thread tries to execute the block, it must wait until the current thread has left the block. This protection is carried out in *Python* using a **lock**. A lock is an instance of the class *Lock* and has two states, *locked* and *unlocked*, as well as two methods *acquire()* and *release()* with the following rules:

state	call	subsequent state/activity
unlocked	acquire()	locked
locked	acquire()	blocked until another thread calls release()
unlocked	release()	error message (RuntimeException)
locked	release()	unlocked

We say that a thread *acquires* the lock with *acquire()* and *releases* it again with *release* [**more...**].

Specifically, to protect a critical block you proceed as follows: First you create a global lock object using *lock = Lock()* that is of course *unlocked* in its initial state. Every thread then tries to acquire the lock using *acquire()* when entering the critical block. If this fails because the lock is already taken, the thread is automatically placed in a waiting state until the lock is free again. If a thread does indeed acquire the lock, it runs through the critical block and then release the lock again when done using *release()* so that the other threads can obtain it [**more...**].

If you understand the passing of the critical block as a resource in a room with a closed door, you can imagine a lock as similar a key, which a thread needs in order to open the door of the room. Upon entry, it takes the key with it and closes the door from the inside. All threads that now want to enter the room must wait for a key in a line in front of the door. Once the thread has done its work in the room, it leaves and closes the door and hangs up the key. The first thread waiting in line takes the key and can now open the door to the room. If no threads are waiting in line, the key remains suspended until a new incoming thread needs it.



In your program, the critical block consists of the drawing and deleting of filled squares. When deleted, the square is painted over with the white background color. The main thread creates a flashing square by drawing the solid red square and then deleting it again after a certain waiting period. In a second thread *MyThread*, the keyboard is continuously prompted with *getKeyCode()*. If the user presses the spacebar, the blinking square is shifted to a random position.

It is self-evident that the critical block needs to be protected by a lock. If the shifting of the square takes place while it is still drawn or deleted, the result is a chaotic behavior.

```
from gpanel import *
from threading import Thread, Lock
import random

class MyThread(Thread):
    def run(self):
        while not isDisposed():
            if getKeyCode() == 32:
                print "----------- Lock requested by MyThread"
                lock.acquire()
                print "----------- Lock acquired by MyThread"
                move(random.randint(2, 8), random.randint(2, 8))
                delay(500)  # for demonstration purposes
                print "----------- Lock releasing by MyThread..."
                lock.release()
            else:
                delay(1)

def square():
    print "Lock requested by main"
    lock.acquire()
    print "Lock acquired by main"
    setColor("red")
    fillRectangle(2, 2)
    delay(1000)
    setColor("white")
    fillRectangle(2, 2)
    delay(1000)
    print "Lock releasing by main..."
    lock.release()

lock = Lock()
makeGPanel(0, 10, 0, 10)

t = MyThread()
t.start()
move(5, 5)
```

```
while not isDisposed():
    square()
    delay(1) # Give up thread for a short while
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

You can track in the console how each thread obediently waits until the lock is releasede:

```
Lock requested by main
Lock acquired by main
---------- Lock requested by MyThread
Lock releasing by main...
---------- Lock acquired by MyThread
Lock requested by main
---------- Lock releasing by MyThread...
Lock acquired by main
```

If you deactivate the lock by commenting out, you will see that the squares are no longer correctly drawn and deleted.

Also note that you should always install a small waiting time in short loops in order to avoid consuming unnecessary processing time.

GUI-WORKERS

Callbacks that are triggered by GUI components run in a specific native thread (sometimes called Event Dispatch Thread (EDT)). This is responsible for ensuring that the entire graphics window along with all components (buttons, etc.) are correctly rendered on the screen. Since the rendering takes place at the end of the callback, the GUI appears to be frozen until the callback returns. This is why no graphical animations are possible in a GUI callback. You must strictly adhere to the following rule:

GUI callbacks must return quickly, i.e. no lengthy operations should be performed in GUI callbacks.

In this case, lengthy means a period of time longer than 10 ms. When dealing with this, you have to assume the worst possible case, i.e. a slow hardware and a heavy system load. If an action lasts longer, you should execute it in a separate thread called GUI worker.

You can draw a Rhodonea rose in your program by clicking on one of the two buttons. The drawing is animated and takes a certain amount of time. Therefore, you have to execute the drawing in a worker thread, which is not a problem with your current knowledge about threading.

There is yet another problem to consider: Since each click of the button creates a new thread, more drawings can be started shortly after one another, leading to chaos. You can prevent this by making the buttons *gray* (*inactive*) during the execution of the drawing.

```
from gpanel import *
from javax.swing import *
import math
import thread

def rho(phi):
    return math.sin(n * phi)

def onButtonClick(e):
```

```
    global n
    enableGui(False)
    if e.getSource() == btn1:
        n = math.e
    elif e.getSource() == btn2:
        n = math.pi
#    drawRhodonea()
    thread.start_new_thread(drawRhodonea, ())

def drawRhodonea():
    clear()
    phi = 0
    while phi < nbTurns * math.pi:
        r = rho(phi)
        x = r * math.cos(phi)
        y = r * math.sin(phi)
        if phi == 0:
          move(x, y)
        else:
          draw(x, y)
        phi += dphi
    enableGui(True)

def enableGui(enable):
    btn1.setEnabled(enable)
    btn2.setEnabled(enable)

dphi = 0.01
nbTurns = 100
makeGPanel(-1.2, 1.2, -1.2, 1.2)
btn1 = JButton("Go (e)", actionListener = onButtonClick)
btn2 = JButton("Go (pi)", actionListener = onButtonClick)
addComponent(btn1)
addComponent(btn2)
validate()
```

MEMO

Only short lasting code should be executed in GUI callbacks, otherwise the graphics system freezes. You have to outsource longer lasting code (more than 10 ms) in a separate worker thread. At any moment, only the components of a GUI whose operation is allowed and useful may be active.

ADDITIONAL MATERIAL

RACE CONDITIONS, DEADLOCKS

Human beings work in a highly parallel way, but their logical reasoning is largely sequential. Because of this, it is difficult for us humans to retain an overview of programs with multiple threads. That is why using threads should be considered carefully, as elegant they may seem at first glance.

Besides applications based on random data, a program should always return the same results (postconditions) with the same initial conditions (preconditions). This is in no way guaranteed for programs with multiple threads accessing shared data, even if the critical areas are protected with locks. You have two threads in your program, *thread1* and *thread2*, carrying out an addition and a multiplication with two global numbers *a* and *b*. *a* and *b* are protected by *lock_a* and *lock_b*. Generate and start both threads one after another in the main part and then wait until they reach the end. Finally, write out the values of *a* and *b*. To generate threads in this example, you use a slightly different notation where you specify the method *run()* as a named parameter in the constructor of the class *Thread*.

```
from threading import Thread, Lock
from time import sleep

def run1():
    global a, b
    print "----------- lock_a requested by thread1"
    lock_a.acquire()
    print "----------- lock_a acquired by thread1"
    a += 5
#    sleep(1)
    print "----------- lock_b requested by thread1"
    lock_b.acquire()
    print "----------- lock_b acquired by thread1"
    b += 7
    print "----------- lock_a releasing by thread1"
    lock_a.release()
    print "----------- lock_b releasing by thread1"
    lock_b.release()

def run2():
    global a, b
    print "lock_b requested by thread2"
    lock_b.acquire()
    print "lock_b acquired by thread2"
    b *= 3
#    sleep(1)
    print "lock_a requested by thread2"
    lock_a.acquire()
    print "lock_a acquired by thread2"
    a *= 2
    print "lock_b releasing by thread2"
    lock_b.release()
    print "lock_a releasing by thread2"
    lock_a.release()

a = 100
b = 200
lock_a = Lock()
lock_b = Lock()

thread1 = Thread(target = run1)
thread1.start()
thread2 = Thread(target = run2)
thread2.start()
thread1.join()
thread2.join()
print "Result: a =", a, ", b =", b
```

MEMO

If you let the program run several times, it sometimes returns the results a = 205, b = 607 and other times a = 210, b = 621. How is this possible? The explanation is as follows: Although *thread1* is created and started in the main part before *thread2*, it is not certain which thread actually begins the execution first. As the first line

```
lock_a requested by thread1
```

or

```
lock_b requested by thread2
```

can be written out. The following course of events is not unique either, since the thread switch can happen anywhere. It could be that the addition or multiplication is performed with the numbers *a* and *b* first, which explains the varying results. Since both threads run together

almost as if in a competition, a **race condition** arises.

However, it could be even worse because the program could also completely freeze. Before the "death" the following is written out:

```
---------- lock_a requested by thread1
lock_b requested by thread2
lock_b acquired by thread2
lock_a requested by thread2
---------- lock_a acquired by thread1
---------- lock_b requested by thread1
```

It takes a bit of detective work to find out what happened there. We will try: Apparently, the *thread1* begins to run first and tries to get *lock_a*. Before it can write out the receipt, *thread2* tries to get *lock_b* and gets this lock. Immediately afterwards, *thread2* also tries to get *lock_a*, but apparently fails because in the meantime *thread1* got it. *thread2* therefore blocks. *thread1* continues to run and tries to get *lock_b,* but also fails because *thread2* has not yet returned it. This also blocks *thread1* and therefore the entire program. Aptly, this situation is called a **deadlock**. (If you activate the two commented out lines using *sleep(1)*, it always results in a deadlock. Think about why this is.)

As you can see, deadlocks occur when two threads *thread1* and *thread2* depend on two shared resources *a* and *b* and block them individually. As a consequence, it can happen that *thread2* waits on *lock_a* and *thread1* waits on *lock_b* and thus both are blocked, with the result that the locks are never released again.

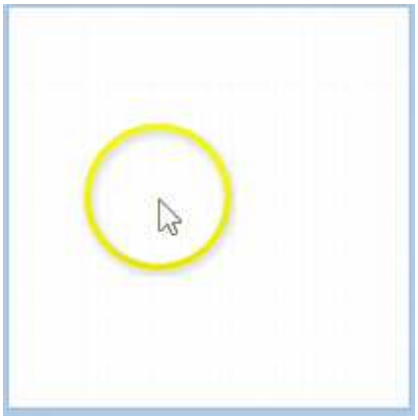
To avoid deadlocks, adhere to the following rule:

Shared resources should be protected with a single lock whenever possible. In addition, it must be ensured that the lock is given back again.

THREAD-SAFE AND ATOMIC EXPRESSIONS

If there are several threads involved, you never know as a programmer the exact point in time or at which spot of the code thread switching occurs. As you have already seen before, it can lead to unexpected and incorrect behavior if the threads work with shared resources. This is especially the case if multiple threads change a window. So if you generate a worker thread in a callback in order to execute long-running code, you should almost always expect that this will result in chaos. In the previous program you avoided this by disabling the buttons during the callback. You can ensure that multiple threads can run the same code without getting in each others way by taking special precautions. Such code is called **thread-safe**. There is an art to writing thread-safe code so that it can be used in an environment with multiple threads without any conflicts [**more..**].

There are few thread-safe libraries, since they are usually less performative and they have the risk of deadlocks. As you experienced above, the *GPanel* library is not thread-safe, whereas the turtle graphics is. You can move several turtles with multiple threads virtually simultaneously. In your program at each mouse click a new thread is created and a new turtle appears at the mouse position. The turtle autonomously draws a star and then fills it.



```
from gturtle import *
import thread

def onMousePressed(event):
#    createStar(event)
    thread.start_new_thread(createStar, (event,))

def createStar(event):
    t = Turtle(tf)
    x = t.toTurtleX(event.getX())
    y = t.toTurtleY(event.getY())
    t.setPos(x, y)
    t.startPath()
    repeat 9:
        t.forward(100)
        t.right(160)
    t.fillPath()

tf = TurtleFrame(mousePressed = onMousePressed)
tf.setTitle("Klick To Create A Working Turtle")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

If you are not generating a new thread (commented out line), you will only see the already finished stars. However, you can write the program without its own thread if you use the named parameter *mouseHit* instead of *mousePressed*, just like you did in **chapter 2.11**. In this case, the thread will be automatically generated in the turtle library.

It is important that you know that switching threads can even be done in the middle of one line of code. For example, even in the line a = a + 1 or a += 1 a thread switch can occur between the reading and writing of the variable value.

In contrast, an expression is called **atomic**, if it can not be interrupted. Similar to most other programming languages, *Python* is also not really atomic. For example, it may happen that a print command is interrupted by print commands of other threads, resulting in a chaotic expression. It is the programmer's task to make functions, expressions, and parts of the code thread-safe and atomic by using locks.

EXERCISES

1. Use a single lock in the above program and set it up so that there are no race conditions and no deadlocks.

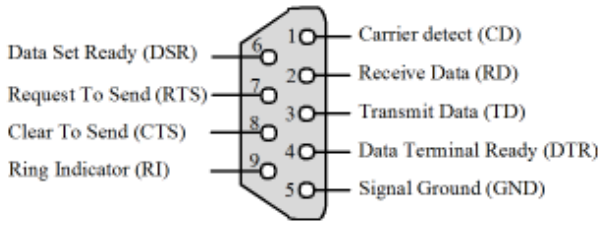
11.5 SERIAL INTERFACE

INTRODUCTION

Although Bluetooth, Ethernet, or USB interfaces are commonly used for communication between a computer and peripherals, the communication over the serial interface (RS-232C) is still widespread since the complexity of the circuit in external devices is lower. That is why the serial interface is still used to connect measuring devices (voltmeters, cathode ray oscilloscopes, etc.) to control devices and robots, and also to communicate with micro-controllers. Modern computers no longer have serial ports, however this problem can be easily solved with low cost USB-to-serial serial adapters.

To understand the serial interface it is important to know that there are data lines for sending and receiving data (TD/RD), two pairs of handshake lines RTS/CTS and DTR/DSR, two status lines CD/RI and a ground. You can see the output lines TD, RTS, DTR and the input lines RD, CTS, DSR, CD, RI from the computer. RTS and DTR can thus be activated and deactivated controlled by the program, and CTS, DSR, CD, and RI can only be read.

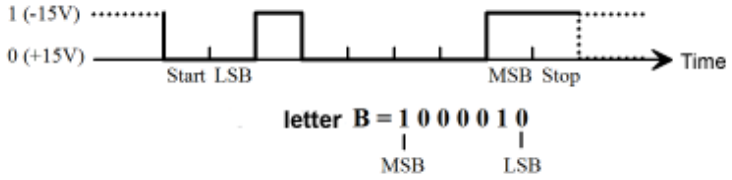
Connections of the 9-pin RS-232 connector:



The format of the transmitted data is simple. It consists of chronologically transmitted data bytes. The transfer begins with a start bit, where the receiver calls attention to the pending data transfer. Then the data itself follows, including 5, 6, 7 or (usually) 8 bits. In order to facilitate an error correction, this is usually followed by a parity bit which indicates whether an odd or an even number of bits were set, but the parity bit may also be omitted. The transfer is completed with one or two stop bits. The sending and receiving devices are not synchronized with each other, i.e. the data transfer can begin and end at any time. However, it is necessary that both devices agree upon the same time duration of a single bit. This is specified by the baud rate (in baud, bits/s) and can usually only be any of the standardized values 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 baud. In addition, both devices can agree on a handshake (flow control), with which they inform each other whether they are ready for the data transfer. You can distinguish between a hardware and a software handshake depending on whether the handshake uses specific handshake pathways or whether it occurs using special ASCII characters (XON/XOFF) embedded in the data stream.

A typical port configuration therefore comprises of: Baud rate, the number of data bits, the number of stop bits, parity none, odd, or even, handshake none, hardware or software.

The flow of voltage during the transmission of the letter 'B', with the configuration *7* databit/no parity/1 stopbit looks thus as follows.



INSTALLATION

The use of the module *pySerial* described here works under a 32- or 64-bit operating system, however only with a 32-bit-Java Runtime Environment (JRE). You can download this from **here**. The following additional installation steps are necessary (*Lib* is a subdirectory of the directory where *tigerjython2.jar* is located):

1. **Download** *pyserial.zip*. Unpack and copy the entire directory structure in the directory *Lib.*

   ```
   <tigerjythonhome>
     Lib
       Serial
         tools
         urhandler
   ```

2. **Download** *javacomm.zip*. Unpack and copy *comm.jar* in *Lib*.

3. (On Windows): Copy the file *win32com.dll from javacomm.zip* in *c.\windows\system32*. In *c:\Program Files (x86)* you will find the directory *Java* and the home directory of the JRE. Copy the file *javax.com.properties from javacomm.zip* to the subdirectory *Lib*. (Note: When updating the JRE, this file may get lost.)

4. If necessary, install the driver of the USB serial adapter that you are using. Connect the adapter and find or reset the used COM port in the adapter properties (in the device manager), e.g. COM1.

5. Test the installation by running *PortEnumerator.jar* (from *javacomm.zip*). The COM port must be displayed.

SIMPLE TERMINAL

For the time being, you should look around in the documentation of *pySerial*. On **http://pyserial.sourceforge.net** under *pySerial API,* you will find a complete description of the classes. However, some of them are platform specific. With your program, you can send characters that you enter on the keyboard one at a time to an external device and write out characters you receive back in a console. It is the simplest form of a terminal emulator.

To test it out, you have two options. You can either connect two computers via a link cable which swaps RD (Receive Data) and TD (Transmit Data) and execute the program run on both, or you can just join these two pins together (hot-connect them with a clip or a similar object). Then all sent characters will be immediately received again.

```
import serial
from gconsole import *

makeConsole()
setTitle("Terminal")
ser = serial.Serial(port = "COM1", baudrate = 2400, timeout = 0)
while not isDisposed():
    delay(1)
    ch = getKey()
    if ch != KeyEvent.CHAR_UNDEFINED:  # a key is typed
        ser.write(ch)
    nbChars = ser.inWaiting()
    if nbChars > 0:
        text = ser.read(nbChars)
        for ch in text:
            if ch == '\n':
                gprintln()
```

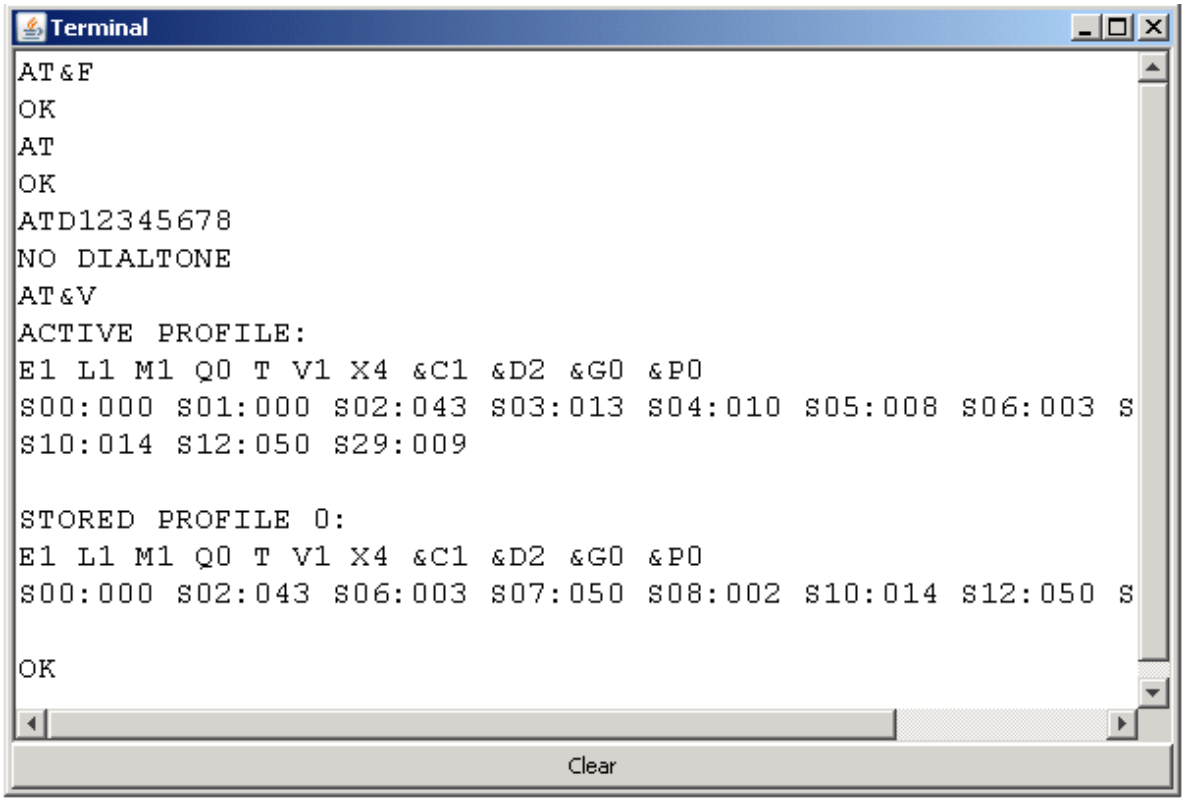
```
            else:
                gprint(ch)
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

MEMO

In order to read the received characters you have to use a non-blocking function, since the program has to constantly check if a key has been pressed. The method *ser.read()* does not block if you set the timeout parameter in the constructor to 0.

If you have a notebook with a built-in modem, the terminal program can communicate with it using the Hayes command set.



11.6. TCP SOCKETS

INTRODUCTION

The exchange of data between computer systems plays an extremely important role in our interconnected world. Therefore we often speak of the combined computer- and communication technologies that should be mastered. In this chapter you learn how to handle the data exchange between two computer systems using the TCP/IP protocol which is used in all Internet connections, for example the Web and all streaming services (data clouds, voice, music, video transmission).

TCPCOM: AN EVENT-DRIVEN SOCKET LIBRARY

The socket programming is based on the client-server model, which has already been described in **Section 6.2**. The server and the client programs are not completely symmetrical. In particular, first the server program must be started before the client program can connect to the server. In order to identify the server on the Internet, its IP address is used. In addition, the server has 65536 communication channels (IP ports), that is selected with a number in the range 0..64535.

When the server starts, it creates a server socket (like a electrical plug) that uses a particular port and goes in a wait state. We say that the server is listening for an incoming client, so the server is in the LISTENING state. The client creates a client socket (the plug counterpart) and tries to establish a communication link to the server using its IP address and port number. The library *tcpcom* simplifies the socket programming essentially, since it describes the current state of the server and client with state variables. The change of the variable is considered to be caused by an event. This programming model corresponds to the natural feeling of many people to describe the communication between two partners by a sequence of events.

As usual in a event-driven model, a callback function, here called *stateChanged(state, msg)* is invoked by the system, when an event is fired. The Python module is integrated into TigerJython, but can also be downloaded from **here** to be studied or used outside TigerJython.

The server is started by creating a *TCPServer* object specifying the ports and the callback function *onStateChanged()* and embarks in the LISTENING state.

```
from tcpcom import TCPServer
server = TCPServer(port, stateChanged = onStateChanged)
```

The callback *onStateChanged (state, msg)* has two string parameters *state* and *msg* that describe the status change of the server:

state	msg	description
TCPServer.LISTENING	port	An existing connection was terminated (or the server is started) and the server listens for a new connection
Server.PORT_IN_USE	port	The server cannot go into the LISTENING state because the port is occupied by another process
TCPServer.CONNECTED	IP address client	A client has signed up and was accepted
TCPServer.MESSAGE	received message	The server has received a message

TCPSever.TERMINATED	(empty)	The server is terminated and does not listen anymore

The client starts with the creation of a *TCPClient* object specifying the IP address of the server, the port and the callback function *onStateChanged ()*. By invoking *connect()* it starts a connection trial.

```
from tcpcom import TCPClient
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```

Again, the callback *onStateChanged (state, msg)* has two string parameters *state* and *msg*, describing the state change of the client:

state	**msg**	**description**
TCPClient.CONNECTING	IP address server	Starting connection attempt
TCPClient.CONNECTION_FAILED	IP address server	Connection trial failed
TCPClient.CONNECTED	IP address server	Connection established
TCPClient.MESSAGE	received message	The client has received a message
TCPClient.DISCONNECTED	(empty)	Connection interrupted (aborted by client or server)

The call to *connect()* is blocking, which means that the function returns *True* once the connection has succeeded, or *False* after a certain timeout period (approximately 10 seconds), if the connection fails. The information about the success or failure of the connection can also be detected via the callback.

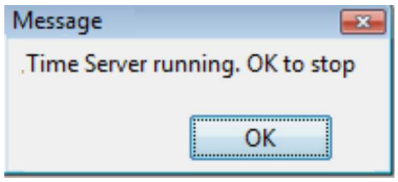
You can try out the client-server programs on the same PC by starting two TigerJython windows. In this case you choose the host address *localhost*. Using two different computers for the client and the server is more close to reality. They must be connected with a network cable or via wireless LAN and the link must be open for TCP/IP communication with the selected port. If the connection fails with your normal hotspot (WLAN access point), this is mostly due to firewall restrictions. In this case you can use your own router or start a mobile hot spot app on your smartphone. Access of the mobile phone to the Internet is not necessary.

As your first socket programming duty you create a server that provides a time service. When a client logs on, it sends the current time (with date) back to the client. There are numerous such time server on the Internet and you can be proud that you are already able to code a professional server application.

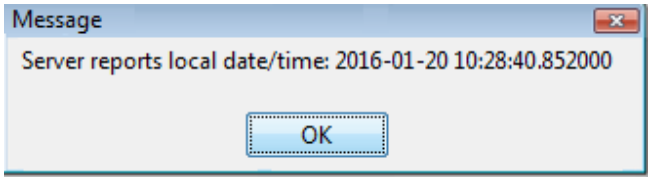
In order to turn off the time server, you use a well-known trick: The main program "hangs" in a TigerJython modal message dialog opened by the blocking function *msgDlg()*. When the function returns by pressing the *OK* or clicking the close button, the server is stopped by calling *terminate()*.

```
from tcpcom import TCPServer
import datetime

def onStateChanged(state, msg):
    print state, msg
    if state == TCPServer.CONNECTED:
        server.sendMessage(str(datetime.datetime.now()))
        server.disconnect()

port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
msgDlg("Time Server running. OK to stop")
server.terminate()
```

The client first asks for the IP address and tries to establish the link to the server by calling *connect()*. The time information received back from the server is written in a dialog window.




```
from tcpcom import TCPClient

def onStateChanged(state, msg):
    print state, msg
    if state == TCPClient.MESSAGE:
        msgDlg("Server reports local date/time: " + msg)
    if state == TCPClient.CONNECTION_FAILED:
        msgDlg("Server " + host + " not available")

host = inputString("Time Server IP Address?")
port = 5000
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```



MEMO

The server and the client implement the event model with a callback function *onStateChanged(state, msg)*. The two parameters provide important information about the event. Make sure that you finish the server with *terminate(),* in order to release the IP port.

ECHO-SERVER

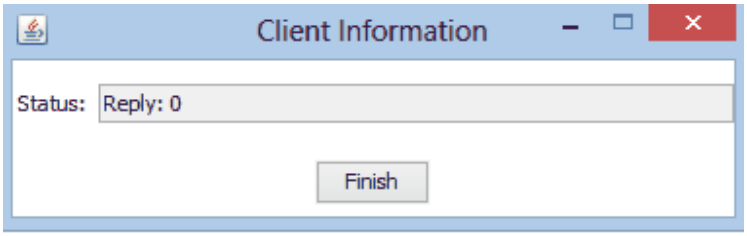
The next training session is famous because it shows the archetype of a client-server communication. The server makes nothing else than sending back the non-modified message received from the client. For this reason it is called an echo server. The system can be easily extended so that the server analyzes the message received from the client and returns a tailored response. This is exactly the concept of all WEB servers, because they reply to a HTTP request issued by the client browser.

You first code the echo server and start it immediately. As you see, the code differs only slightly from the time server. For illustration and debugging purposes you display the *state* and *msg* parameter values in the console window.

```
from tcpcom import TCPServer

def onStateChanged(state, msg):
    print state, msg
    if state == TCPServer.MESSAGE:
        server.sendMessage(msg)

port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
msgDlg("Echo Server running. OK to stop")
server.terminate()
```

For the client you invest a bit more effort by using the *EntryDialog* class to display a non-modal dialog to show status information. Also in other contexts the *EntryDialog* is very convenient, since you can easily add editable and non-editable text fields and other GUI elements such as various types of buttons and sliders.

```
from tcpcom import TCPClient
from entrydialog import *
import time

def onStateChanged(state, msg):
    print state, msg
    if state == TCPClient.MESSAGE:
        status.setValue("Reply: " + msg)
    if state == TCPClient.DISCONNECTED:
        status.setValue("Server died")

def showStatusDialog():
    global dlg, btn, status
    status = StringEntry("Status: ")
    status.setEditable(False)
    pane1 = EntryPane(status)
    btn = ButtonEntry("Finish")
    pane2 = EntryPane(btn)
    dlg = EntryDialog(pane1, pane2)
    dlg.setTitle("Client Information")
    dlg.show()

host = "localhost"
port = 5000
showStatusDialog()
client = TCPClient(host, port, stateChanged = onStateChanged)
status.setValue("Trying to connect to " + host + ":" + str(port) + "...")
time.sleep(2)
rc = client.connect()
if rc:
    time.sleep(2)
    n = 0
    while not dlg.isDisposed():
        if client.isConnected():
            status.setValue("Sending: " + str(n))
            time.sleep(0.5)
            client.sendMessage(str(n), 5)  # block for max 5 s
            n += 1
        if btn.isTouched():
            dlg.dispose()
        time.sleep(0.5)
    client.disconnect()
else:
    status.setValue("Connection failed.")
```



MEMO

In the client program you use the function *sendMessage(msg, timeout)* with an additional *timeout* parameter. The call is blocking for a maximum time interval (in seconds) until the server sends back a response. *sendMessage()* returns either the server's response or None, if no response is received within the given timeout.

It is important to know the difference between modal and non-modal (modeless) dialog boxes. While the modal window blocks the program until the dialog is closed, with a modeless dialog the program continues, so that at any time status information can be displayed and user input read by the running program.

TWO PERSONS ONLINE GAME WITH TURTLE GRAPHICS

"Sinking Boats" is a popular game between two people, in which the memory plays an important role. The playing boards of each player is a one or two dimensional arrangement of grid cells where vessels are placed. They occupy one or multiple cells and have different values depending on the type of the ship. By turns, each player denotes a cell where, so to speak, a bomb is dropped. If the cell contains a portion of a ship, it will be sunk, so removed from the board and its value credited to the aggressor. If one of the players has no more ships, the game ends and the player with the greater credit wins.

In the simplest version, the ships are displayed as colored square cells in a one-dimensional array. All ships are of equal rank. The winner is the player who has first eliminated all enemy ships.

As for the game logic, the client and server programs are largely identical. To show the board, it is sufficient to use elementary drawing operations from turtle graphics that you know for a long time. You select a cell numbering from 1 to 10 and you have to produce 4 different random numbers out of 1..10 for the random selection of the ships. To do so, is is elegant to use the library function *random.sample()*. To launch a bomb, you send the full Python command to the partner who executes it with *exec()*. (Such a dynamic code execution is only possible in a few programming languages.) To find out whether there has been a hit, you test the background color with *getPixelColorStr()*.

In the game, the two players are equal, but their programs vary slightly, depending on who is server and and who is client. In addition, the server must start first.

```
from gturtle import *
from tcpcom import TCPServer
import random

def initGame():
    clear("white")
    for x in range(-250, 250, 50):
        setPos(x, 0)
        setFillColor("gray")
        startPath()
        repeat 4:
            forward(50)
            right(90)
```

```
        fillPath()

def createShips():
    setFillColor("red")
    li = random.sample(range(1, 10), 4) # 4 unique random numbers
    for i in li:
        fill(-275 + i * 50, 25)

def onMouseHit(x, y):
    global isMyTurn
    setPos(x, y)
    if getPixelColorStr() == "white" or isOver or not isMyTurn:
        return
    server.sendMessage("setPos(" + str(x) + "," + str(y) + ")")
    isMyTurn = False

def onCloseClicked():
    server.terminate()
    dispose()

def onStateChanged(state, msg):
    global isMyTurn, myHits, partnerHits
    if state == TCPServer.LISTENING:
        setStatusText("Waiting for game partner...")
        initGame()
    if state == TCPServer.CONNECTED:
        setStatusText("Partner entered my game room")
        createShips()
    if state == TCPServer.MESSAGE:
        if msg == "hit":
            myHits += 1
            setStatusText("Hit! Partner's remaining fleet size "
                + str(4 - myHits))
            if myHits == 4:
                setStatusText("Game over, You won!")
                isOver = True
        elif msg == "miss":
            setStatusText("Miss! Partner's remaining fleet size "
                + str(4 - myHits))
        else:
            exec(msg)
            if getPixelColorStr() != "gray":
                server.sendMessage("hit")
                setFillColor("gray")
                fill()
                partnerHits += 1
                if partnerHits == 4:
                    setStatusText("Game over, Play partner won!")
                    isOver = True
                    return
            else:
                server.sendMessage("miss")
            setStatusText("Make your move")
            isMyTurn = True

makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)
addStatusBar(30)
hideTurtle()
port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
isOver = False
isMyTurn = False
myHits = 0
partnerHits = 0
```

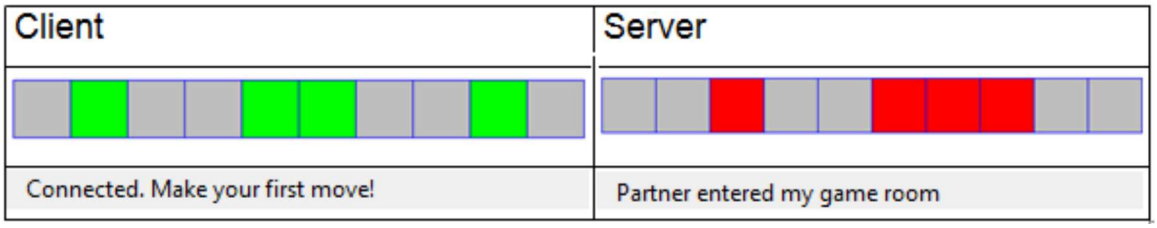
Highlight program code (Ctrl+C copy, Ctrl+V paste)

The client program is almost the same:

```
from gturtle import *
import random
from tcpcom import TCPClient

def initGame():
    for x in range(-250, 250, 50):
        setPos(x, 0)
        setFillColor("gray")
        startPath()
        repeat 4:
            forward(50)
            right(90)
        fillPath()

def createShips():
    setFillColor("green")
    li = random.sample(range(1, 10), 4) # 4 unique random numbers 1..10
    for i in li:
        fill(-275 + i * 50, 25)

def onMouseHit(x, y):
    global isMyTurn
    setPos(x, y)
    if getPixelColorStr() == "white" or isOver or not isMyTurn:
        return
    client.sendMessage("setPos(" + str(x) + "," + str(y) + ")")
    isMyTurn = False

def onCloseClicked():
    client.disconnect()
    dispose()

def onStateChanged(state, msg):
    global isMyTurn, myHits, partnerHits
    if state == TCPClient.DISCONNECTED:
        setStatusText("Partner disappeared")
        initGame()
    elif state == TCPClient.MESSAGE:
        if msg == "hit":
            myHits += 1
            setStatusText("Hit! Partner's remaining fleet size "
                + str(4 - myHits))
            if myHits == 4:
                setStatusText("Game over, You won!")
                isOver = True
        elif msg == "miss":
            setStatusText("Miss! Partner's remaining fleet size "
                + str(4 - myHits))
        else:
            exec(msg)
            if getPixelColorStr() != "gray":
                client.sendMessage("hit")
                setFillColor("gray")
                fill()
                partnerHits += 1
                if partnerHits == 4:
                    setStatusText("Game over, Play partner won")
                    isOver = True
                    return
            else:
                client.sendMessage("miss")
            setStatusText("Make your move")
            isMyTurn = True

makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)
addStatusBar(30)
hideTurtle()
```

```
initGame()
host = "localhost"
port = 5000
client = TCPClient(host, port, stateChanged = onStateChanged)
setStatusText("Client connecting...")
isOver = False
myHits = 0
partnerHits = 0
if client.connect():
    setStatusText("Connected. Make your first move!")
    createShips()
    isMyTurn = True
else:
    setStatusText("Server game room closed")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)



MEMO

In two players games, the players can have the same or different game situations. In the second case, which includes most card games, the game must necessarily be played with two computers, because the game situation must be kept secret.

Instead of always let the client start the game, the player to move first could be selected randomly or by negotiating.

Both programs are completed by clicking on the close button in the title bar. But then some additional "cleanup" is required like stopping the server or breaking the communication link. To do so, register the callback *onCloseClicked()* and the default closing process is disabled. It is now up to you, to perform the adequate operation in the callback and closing the turtle window by calling *dispose()*. Disregarding so, the Task Manager is the needed to get you out of a jam.

TWO PERSONS ONLINE GAME WITH GAMEGRID

For more complex games, it is of great advantage to refer to a game library that simplifies the code considerably. You already learned in **Chapter 7** how to use *GameGrid*, a full-featured game engine integrated into TigerJython. By combining *GameGrid* with *tcpcom*, you can create sophisticated multi-person online games where the game partners are located anywhere in the world. For illustration you expand the "Sinking Boats" game in 2 dimensions. The game logic remains unchanged, however you transfer now x and y coordinates of the selected cell. On the receiving side, the partner can find out if one of his ships is hit and report back a "hit" or "miss" message.

In game programming it is important to make a special effort that games proceed "reasonably", even if the two players behave somewhat unreasonably. So the firing of bombs must be disabled if it is not the player's move. When one of the programs terminates unexpectedly, the partner should be informed. Although these additional checks blow up the code, this is the hallmark of careful programming

Since a large part of the code for the server and client is the same, and code duplication is one of the greatest sins of programming, the common code is exported into a module *shiplib.py* that can imported by both applications. Different behavior is taken into account by additional parameters like *node* that refers to a *TCPServer* or *TCPClient* object.

Library:

```
# shiplib.py

from gamegrid import *

isOver = False
isMyMove = False
dropLoc = None
myHits = 0
partnerHits = 0
nbShips = 2

class Ship(Actor):
     def __init__(self):
         Actor.__init__(self, "sprites/boat.gif")

def handleMousePress(node, loc):
    global isMyMove, dropLoc
    dropLoc = loc
    if not isMyMove or isOver:
          return
    node.sendMessage("" + str(dropLoc.x) + str(dropLoc.y)) # send location
    setStatusText("Bomb fired. Wait for result...")
    isMyMove = False

def handleMessage(node, state, msg):
    global isMyMove, myHits, partnerHits, first, isOver
    if msg == "hit":
        myHits += 1
        setStatusText("Hit! Partner's fleet size " + str(nbShips - myHits)
            + ". Wait for partner's move!")
        addActor(Actor("sprites/checkgreen.gif"), dropLoc)
        if myHits == nbShips:
            setStatusText("Game over, You won!")
            isOver = True
    elif msg == "miss":
        setStatusText("Miss! Partner's fleet size " + str(nbShips - myHits)
            + ". Wait for partner's move!")
        addActor(Actor("sprites/checkred.gif"), dropLoc)
    else:
        x = int(msg[0])
        y = int(msg[1])
        loc = Location(x, y)
        bomb = Actor("sprites/explosion.gif")
        addActor(bomb, loc)
        delay(2000)
        bomb.removeSelf()
        refresh()
        actor = getOneActorAt(loc, Ship)
        if actor != None:
            actor.removeSelf()
            refresh()
            node.sendMessage("hit")
            partnerHits += 1
            if partnerHits == nbShips:
                setStatusText("Game over! Partner won")
                isOver = True
                return
        else:
            node.sendMessage("miss")
        isMyMove = True
        setStatusText("You fire!")
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

By using the external module *shiplib.py* the code of the server and the client becomes much simpler and clearer.

Server:

```
from gamegrid import *
from tcpcom import TCPServer
import shiplib

def onMousePressed(e):
     loc = toLocationInGrid(e.getX(), e.getY())
     shiplib.handleMousePress(server, loc)

def onStateChanged(state, msg):
    global first
    if state == TCPServer.PORT_IN_USE:
        setStatusText("TCP port occupied. Restart IDE.")
    elif state == TCPServer.LISTENING:
        setStatusText("Waiting for a partner to play")
        if first:
            first = False
        else:
            removeAllActors()
            for i in range(shiplib.nbShips):
                addActor(shiplib.Ship(), getRandomEmptyLocation())
    elif state == TCPServer.CONNECTED:
        setStatusText("Client connected. Wait for partner's move!")
    elif state == TCPServer.MESSAGE:
        shiplib.handleMessage(server, state, msg)

def onNotifyExit():
    server.terminate()
    dispose()

makeGameGrid(6, 6, 50, Color.red, False, mousePressed = onMousePressed,
             notifyExit = onNotifyExit)
addStatusBar(30)
for i in range(shiplib.nbShips):
    addActor(shiplib.Ship(), getRandomEmptyLocation())
show()
port = 5000
first = True
server = TCPServer(port, stateChanged = onStateChanged)
shiplib.node = server
```

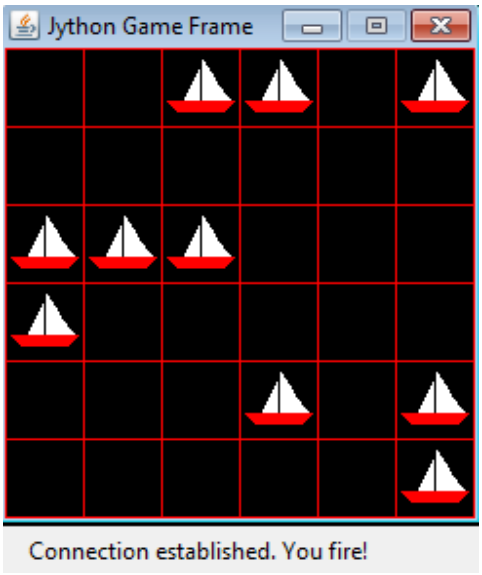
Highlight program code (Ctrl+C copy, Ctrl+V paste)

Client:

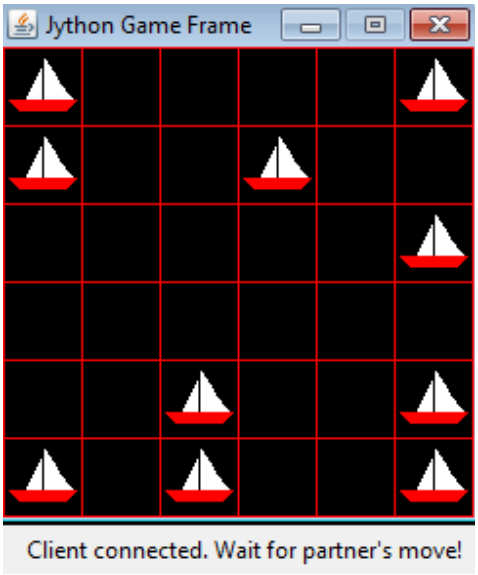
```
from gamegrid import *
from tcpcom import TCPClient
import shiplib

def onMousePressed(e):
     loc = toLocationInGrid(e.getX(), e.getY())
     shiplib.handleMousePress(client, loc)

def onStateChanged(state, msg):
    if state == TCPClient.CONNECTED:
        setStatusText("Connection established. You fire!")
        shiplib.isMyMove = True
    elif state == TCPClient.CONNECTION_FAILED:
        setStatusText("Connection failed")
    elif state == TCPClient.DISCONNECTED:
        setStatusText("Server died")
        shiplib.isMyMove = False
    elif state == TCPClient.MESSAGE:
        shiplib.handleMessage(client, state, msg)

def onNotifyExit():
    client.disconnect()
    dispose()
```

```
makeGameGrid(6, 6, 50, Color.red, False,
    mousePressed = onMousePressed, notifyExit = onNotifyExit)
addStatusBar(30)
for i in range(shiplib.nbShips):
    addActor(shiplib.Ship(), getRandomEmptyLocation())
show()
host = "localhost"
port = 5000
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```

Client



Server



MEMO

The end of the game is a special situation that must be programmed carefully. Since it is an "emergency state", you are using a global flag *isOver* that is set to *True* when the game is over. It also raises the question of whether the game can be played again without restarting the server or client program. In this implementation, the client needs to terminate his program and the server then goes back into the LISTENING state, waiting for a new client to play again.

The programs could be improved in many ways. As general rule game programming is very attractive because there are no limits for the imagination and ingenuity of the developer. Moreover, after all your effort, relaxing and playing the game is also lot of fun.

Until now the two playmates have to start their programs according to the rule "First, the server and then the client". To circumvent this restriction the following trick could be applied: A program always starts first as a client and tries to create a connection to the server. If this fails, it starts a server [**more...**].

COMMUNICATION WITH HANDSHAKE

Even in every days life, communication between two partners need some kind of synchronization. In particular, data may be sent only if the recipient is actually ready for the reception and further processing. Failure to observe this rule may result in data loss or even block the programs. Different computing power of the two nodes and a variing transmission time must be considered too.

One known method to get these timing problems under control, is to provide feedback from the receiver to the transmitter, which can be compared with an amicable handshake. The process is basically simple: data is transmitted in blocks and the receiver acknowledges the correct reception and and readiness for the next block with a feedback. Only when it is received, the next block is sent. The feedback may also cause the transmitter to send the same block again [**more...**].

To demonstrate the handshake principle, your the turtle program of the server draws relatively slowly lines dictated by the client's mouse clicks. The client's turtle moves much faster. Therefore the client must wait from click to click until the server reports back that he has completed its drawing operation and is ready to dispatch the next command.

Server:

```
from gturtle import *
from tcpcom import TCPServer

def onCloseClicked():
    server.terminate()
    dispose()

def onStateChanged(state, msg):
    if state == TCPServer.MESSAGE:
        li = msg.split(",")
        x = float(li[0])
        y = float(li[1])
        moveTo(x, y)
        dot(10)
        server.sendMessage("ok")

makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)
port = 5000
server = TCPServer(port, stateChanged = onStateChanged)
```

Client:

```
from gturtle import *
from tcpcom import TCPClient

def onMouseHit(x, y):
    global isReady
    if not isReady:
        return
    isReady = False
    client.sendMessage(str(x) + "," + str(y))
    moveTo(x, y)
    dot(10)

def onCloseClicked():
    client.disconnect()
    dispose()

def onStateChanged(state, msg):
   global isReady
   if state == TCPClient.MESSAGE:
        isReady = True

makeTurtle(mouseHit = onMouseHit, closeClicked = onCloseClicked)

speed(-1)
host = "localhost"
port = 5000
isReady = True
client = TCPClient(host, port, stateChanged = onStateChanged)
client.connect()
```

MEMO

To bring the actions of the transmitter and receiver in an orderly temporal sequence, the transmitter waits to send the next data until he receives a feedback that the receiver is ready for further processing.

EXERCISES

1a. Create a client-server system in which the server provides good math skills. In an input dialog the client can enter any function from the module *math*, for example s*qrt(2)*, and gets the result as a server response that is displayed in the output window. Use the Python function *exec(cmd)* to execute a given command.

1b. Modify your program that the server reports "illegal" if he cannot fulfill the request. Hint: Catch the exception that is thrown by *exec()*.

2. Create a remote commander: The server starts a turtle window, waiting for a commander client. In a client dialog box, you can type a turtle command that is sent to the server where the command is executed. If successful, "OK", otherwise "Fail" is sent back to the client and displayed there. After receiving this response, the commander can send the next command. Note: You can also merge together multiple commands separated by a semicolon.

3. Add sound output to the two-dimensional Sinking-Boats game. Suggestion: Emit a short sound clip when firing, and when you get a hit or miss feedback. Use your skills from **Chapter 4**: Sound. You can use predefined sounds, but your own sounds are funnier.

4*. Improve the one-dimensional Sinking-Boat game so ships are no more represented by colored cells, but with pictures that disappears when hit.

ADDITIONAL MATERIAL

REMOTE-SENSING WITH RASPBERRY PI

Client-server communications over TCP/IP play an important role in measuring and control technology. Often a measuring instrument or a robot is far away from a command center and the measurement and control data are transmitted via TCP/IP. In example 2 you already realized a remote control. Below you are using a Raspberry Pi, which acts as interface between the measuring sensors and the Internet, and sends the measurement data to a remote recording station. To simplify the system, no actual sensors are used and only the pressed or released state of a pushbutton is reported.

For the Raspberry Pi you write a measurement detection program in Python using the *tcpcom* module that you need to copy to the same directory on the RPi. You can either work with a keyboard and a screen directly attached to the RPi or from a remote PC using SSH, WinSCP or VNC. For further instructions, check the literature or search the Internet.

The server program is very simple: In the measuring loop you call periodically *GPIO.input()* to get the state of the button. Here 0 is reported when the button is pressed. Subsequently, the button state information is transferred to the client. By holding the key pressed during 3 measuring cycles, the server program terminated.

```
import time
import RPi.GPIO as GPIO
```

```
from tcpcom import TCPServer

def onStateChanged(state, msg):
    print "State:", state, "Msg:", msg

P_BUTTON1 = 16 # Switch pin number
dt = 1  # 1 s period
port = 5000 # IP port

GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)
GPIO.setup(P_BUTTON1, GPIO.IN, GPIO.PUD_UP)
server = TCPServer(port, stateChanged = onStateChanged)
n = 0
while True:
    if server.isConnected():
        rc = GPIO.input(P_BUTTON1)
        if rc == 0:
            server.sendMessage("pressed")
            n += 1
            if n == 3:
                break
        else:
            server.sendMessage("released")
            n = 0
        time.sleep(dt)
server.terminate()
print "Server terminated"
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

The client writes only the data obtained in the output window.

```
from tcpcom import TCPClient

def onStateChanged(state, msg):
    print "State: " + state + ". Message: " + msg

host = "192.168.0.5"
#host = inputString("Host Address?")
port = 5000 # IP port
client = TCPClient(host, port, stateChanged = onStateChanged)
rc = client.connect()
if rc:
    msgDlg("Connected. OK to terminate")
    client.disconnect()
```

Highlight program code (Ctrl+C copy, Ctrl+V paste)

LITERATURE & LINKS

- **http://tigerjython.ch/download/tigerjython_en.pdf**
  Download tutorial (PDF)

- **http://examples.tigerjython.ch**
  Source code of all examples

- **http://jython.tobiaskohn.ch/PythonScript.pdf**
  Python. An Introduction to Computer Programming by Tobias Kohn (in German)

- **http://www.jython.ch**
  Turtle Graphics, Robotics and Games by Jarka Arnold (in German)

- **http://www.aplu.ch/jython**
  Libraries and Examples by Aegidius Plüss (in English)

- **http://www.tigerjython.ch/download/ACMandIEreport.pdf**
  Informatics education: Europe cannot afford to miss the boat

- **http://www.tigerjython.ch/download/ForteGuzdialCommNotCalc.pdf**
  Computers for Communication, Not Calculation:
  Media as a Motivation and Context for Learning

- **http://de.padlet.com/myschool/python**
  Introduction to Programming with Python by Günter Öller, Linz (Austria)

- **http://fit-in-it.ch/sites/default/files/downloads/informatik_d.pdf** informatik@gymnasium
  A Proposal for Switzerland by J. Kohlas, J. Schmid, C.A. Zehnder, ed. (Hasler Foundation)

Literature References:

- Böhm C., Jacopini G., *Flow diagrams, turing machines and languages with only two formation rules*, Communications of the ACM 9(5), 366-371 (1966)

- Wirth Niklaus, *Algorithms and Data Structures*, Pearson Education (1985)

- Wong Baoswan Dzung, *Bézierkurven: gezeichnet und gerechnet*, Orell Füssli (2003)

CONTACT

help@tigerjython.com

Developer team: Jarka Arnold, University of Teacher Education Bern
www.java-online.ch

Tobias Kohn, Cantonal School Zürich Oberland
www.tobiaskohn.ch

Dr. Aegidius Plüss, University of Bern
www.aplu.ch

About the authors

Jarka Arnold
Jarka Arnold has longstanding experience as a lecturer of computer science at the Bern University of Teacher Education. Web-based learning environments for programming courses were developed under her guidance in the context of several research projects, which are successfully being used at many education institutions (**http://www.java-online.ch** and **http://www.jython.ch**).

Tobias Kohn
Tobias Kohn (**http://www.tobiaskohn.ch/**) completed his studies of mathematics at the ETH Zürich in 2008 and since then works as a teacher of mathematics and computer science at the Swiss high school (Gymnasium) Zürcher Oberland in Wetzikon. In the fall of 2012 he began his doctoral studies at the ETH Zürich in addition to his teaching, and seeks ways of simplifying the introduction to computer programming.

Aegidius Plüss
Aegidius Plüss (**http://www.aplu.ch**) is a former professor for computer science and its didactics at the University of Bern. He wrote the course book "Java exemplarisch" and was involved in several courses in further education for computer science teachers. He develops extensive libraries and programming environments for computer science classes.