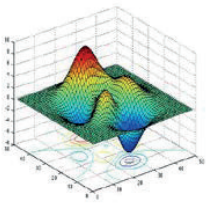
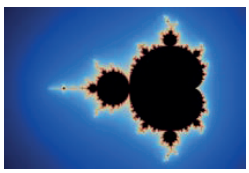
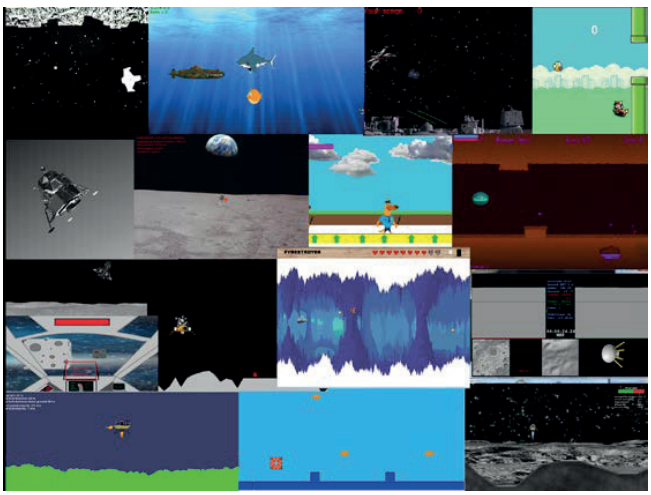
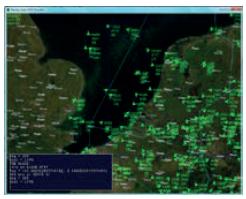
Programming and Scientific Computing in

for Aerospace Engineers









AE Tutorial Programming Python v4.1

Jacco Hoekstra

TU Delft Delft University of Technology

Artikelnummer 06917700057

```
print("Hello world!")
```

```
>>> Hello world!
```

```
if language==python:
    programming = fun
```

Table of Contents

Preface

This first version of this reader was developed for, and during the pilot of, the Programming course in the first year of the BSc program Aerospace Engineering at the Delft University of Technology in 2012. It is still a living document and will be expanded and adapted (and debugged) for another year. This version is the first version for Python 3, as the course has transitioned in 2018 from Python 2 to Python 3.

The goal of the Python programming course is to enable the student to:

- write a program for scientific computing
- develop models
- analyze behavior of the models using e.g. plots
- visualize models by animating graphics

The course assumes some mathematical skills, but no programming experience whatsoever.

This document is provided as a reference for the elaboration of the assignments. The reader is encouraged to read through the relevant chapters applicable to a particular problem. For later reference, many tables, as well as some appendices with quick reference guides, have been included. These encompass the most often used functions and methods. For a complete overview, there is the excellent documentation as provided with Python in the IDLE Help menu, as well as the downloadable and on-line documentation for the Python modules *Numpy, Scipy, Matplotlib* and *Pygame*.

Also, the set-up of the present course is to show the appeal of programming. Having this powerful tool at hand allows the reader to use the computer as a 'mathematical slave'. And by making models one basically has the universe in a sandbox at one's disposal: Any complex problem can be programmed and displayed, from molecular behavior to the motion in a complex gravity field in space.

An important ingredient at the beginning of the course is the ability to solve mathematical puzzles and numerical problems. Together with the easy to use graphics module *Pygame* module has been included in this reader. This allows, next to the simulation of a physical problem, a real-time visualization and some control (mouse and keyboard) for the user, which also adds some fun for the beginning and struggling programmer.

Next to the mathematical puzzles, challenges (like Project Euler and the Python challenge) and simulations and games, there is a programming contest included in the last module of the course for which there is a prize for the winners. Often students surprise me with their skills and creativity in such a contest by submitting impressive simulations and games. In fact the center picture at the cover of this reader is a set of screenshots of the context of 2017.

Many thanks to the students and teaching assistants, whose questions, input and feedback formed the foundation for this reader.

Prof.dr.ir. Jacco M. Hoekstra
Faculty of Aerospace Engineering
Delft University of Technology

1. Getting started

1.1 What is programming?

Ask a random individual what programming is and you will get a variety of answers. Some love it. Some hate it. Some call it mathematics, others philosophy, and making models in Python is mostly a part of physics. More interestingly, many different opinions exist on how a program should be written. Many experienced programmers tend to believe they see the right solution in a flash, while others say it always has to follow a strict phased design process, starting with thoroughly analyzing the requirements (not my style). It definitely is a skill and I think it's also an art. It does not require a lot of knowledge, it is a way of thinking and it becomes an intuition after a lot of experience.

This also means that learning to program is very different from the learning you do in most other courses. In the beginning, there is a very steep learning curve, but once you have taken this first big step, it will become much easier and basically a lot of fun. But how and when you take that first hurdle is very personal. Of course, you need to achieve the right rate of success over failure, something you can achieve by testing small parts during the development. For me, there aren't many things that give me more pleasure than to see my program (finally) work. The instant, often visual, feedback makes it a very rewarding activity.

And even though at some stage you will also see the right solution method in a flash, at the same time your program will almost certainly not work the first time you run it. A lot of time is spent understanding why it will not work and fixing this. Therefore some people call the art of programming: "solving puzzles created by your own stupidity"!

While solving these puzzles, you will learn about logic, you will learn to think about thinking.

The first step towards a program is always to decompose a problem into smaller steps, into ever smaller building blocks to describe the so-called algorithm. An algorithm is a list of actions and decisions that a computer (or a person) has to go through chronologically to solve a problem.

This is often schematically presented in the form of a flow chart. For instance, the algorithm of a thermostat that has to control the room temperature is shown in figure 1.1.

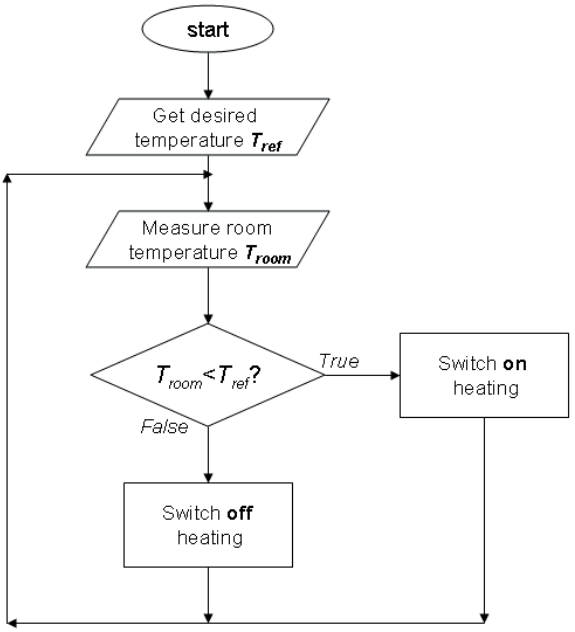


Figure 1.1: Flow chart of 'thermostat' algorithm.

Another way to design and represent algorithms is using simplified natural language. Let's take as an example the algorithm to "find the maximum value of four numbers". We can detail this algorithm as a number of steps:

Let's call the 4 numbers a,b,c and d

if a >b then make x equal to a, else make x equal to b

if c < d then make y equal to c, else make y equal to d

if x < y then make x equal to y

show result x on screen

Going through these steps, the result will always be that the maximum value of the four numbers is shown on the screen. This kind of description in natural language is called "pseudo-code".

This pseudo-code is already very close to how Python looks, as this was one the goals of Python: it should read just as clear as pseudo-code. But before we can look at some real Python code, we need to know what Python is and how you can install it. After that, we will have a look at some simple programs in Python, which you can try out in your freshly installed Python environment.

1.2 What is Python?

Python is a general purpose programming language. And even though until a few years ago Python was used more in the USA than in Europe, it has been developed by a Dutchman, Guido van Rossum. It all started as a hobby project, which he pursued in his spare time while still employed at the so-called Centrum Wiskunde & Informatica (CWI) in Amsterdam in 1990. Python was named after Monty Python and references to Monty Python in comments and examples are still appreciated. The goals of Python, as Guido has formulated them in a 1999 DARPA proposal, are:

- an **easy** and **intuitive** language just as **powerful** as major competitors
- **open source**, so anyone can contribute to its development
- code that is as **understandable** as plain English
- suitable for everyday tasks, allowing for **short development times**

Guido van Rossum was employed by Google for many years, as this is one of the many companies that use Python. He is currently working for another user of Python: Dropbox. He still is the moderator of the language, or as he is called by the Python community: the "benevolent dictator for life".

A practical advantage of Python is that it supports many platforms (Windows, Apples, Linux) and that it is free, and so are all add-ons. Many have been developed by the large (academic) Python community. Some have become standards of their own, such as the scientific package Numpy/Scipy/Matplotlib. These scientific libraries (or modules as they are called in Python), in syntax(grammar) heavily inspired by the software package MATLAB, are now the standard

libraries for scientific computing in Python. IEEE has named Python as the de facto standard programming language for data analysis.

There are currently still two versions around: Python 2 and Python 3. There is also a second and a third number indicating the exact version, but these as less relevant as they are downwards compatible with the other 2.x and 3.x versions. At the time of writing the newest versions was 3.6 and support for Python 2.7 had been terminated.

Up to Python 3 all versions were downwards compatible. So all past programs and libraries will still work in a newer Python 2.x versions. However, at some point, Guido van Rossum, being the BDFL (Benevolent Dictator For Life) of Python, wanted to correct some fundamental issues, which could only be done by breaking the downward compatibility. This started the development of Python 3.0. For a long time Python 2.x was also still maintained and updated. This has stopped in 2017. So now the majority of the community is moving or has moved to Python 3.

An example of a difference between the two versions is the syntax of the PRINT-function, which shows a text or a number on the screen during runtime of the program:

In Python 3.x: `print("Hello World!")`

In Python 2.x: `print "Hello world!"`

(Another major difference applies to **integer division** and the **input( )** function, but we need to know more about data types to understand that.)

Python is called a scripting language. This means that the Python source is interpreted when you run the program as opposed to needing to compile all program files first, then link them together with the correct libraries to make an executable and then run this. This is very user-friendly: there is no need to compile or link files before you can run the same program on any operating system: Windows, Apple's Mac OS or Linux . Python libraries are binary but most are available on all three supported platforms.

The penalty for this is, depending on your program structure, sometimes some execution speed. Some (milli)seconds of runtime are traded for shorter development times, which saves time in the order of days or weeks. Note that Python libraries like *Numpy* and *Scipy* use very fast low-level modules, resulting in extremely fast execution times for scientific computing. By using a similar syntax , it is replacing MATLAB (which also uses a small scripting language) worldwide. Using vectorizes programming and the numpy librarie make runtaimes comparable with optimized C++ and Fortran in many cases for the scientific computing. The same mechanism allows fats graphics. For example for 2D graphics the *Pygame* graphics library, which is a layer over the very fast and famous SDL library used is also extremely fast.

Using an interpreter instead of a compiler, means you need to have Python, with its interpreter, installed on the computer where you run the Python program. Would you ever want to make an executable to avoid this requirement, there are modules, like distutils, or add-ons, like called *Py2exe* creating self-contained application which does not need Python installed. Such an executable will only run on the OS under which it was compiled. How this should be done, is covered by a separate chapter of the reader, called 'Distributing your Python program'. Using the

Inno Setup tool, you can make a installer for your application on Windows for example. This tool converts the complete folder with data and the program into one setup executable file, which is also covered in this chapter.

1.3 Installing Python

There are two ways to install Python with the libraries we need:

1. **Standard manual install**, using pip for the packages (recommended)
2. Download a **package** e.g. Anaconda. This results in a non-standard installation, you may still need to use pip to install not-included packages like the pygame graphics module

1.3.1 Windows

Go to Python.org and download the latest version of Python 3 64-bits. Run the installer, do a full install, so select all options in the installer. If necessary, also select 'pip' and choose the option of 'the path' to be added.

Now you have pip installed, which makes it relatively simple to install the other packages. Pip makes sure to download the exactly right version for your installation of Python and OS and will install it.

Now we install the additional modules which we will use in this course. In the Windows Start Menu, search for **'Command prompt'** and right click it. Then select **Run as Administrator** and type the following commands:

```
pip install scipy matplotlib pygame spyder
```

1.3.2 Linux

Open a terminal, then type follow two lines (replace 'apt' with 'zypper' on for RPM based Linux):

```
sudo apt install python3 python3-pip
pip install numpy scipy matplotlib pygame spyder
```

1.3.3 Apple

Part 1: Homebrew and python3
Mac OSX has a python 2 installation already in the system, which is used for various tasks. Adding python 3 to the system itself might cause trouble later on, therefore we use Homebrew to install programs and python in a separate location, so that this does interfere with the default python installation.

To install homebrew, you need two steps:

- Open a terminal window to interact with your Mac through the command line; go to launchpad, type "terminal" in the search window, and click on the "Terminal" icon.

- Access the Homebrew installation instructions with a browser, at https://brew.sh

The terminal will typically show you a prompt line, like:

```
mymac:~ me$
```

The first word (here "`mymac`") is the name you gave your computer, then it shows that you are currently in the home directory ("`:~`"), and your user name ("`me`"). Any commands typed after the "`$`" prompt are interpreted and executed by the computer when you hit the "Enter" key.

Copy the line for Homebrew installation from the Homebrew web page, and copy it into the terminal. The installation command will ask for your permission to continue, and for your password (you need administrator rights for this step). Let the Homebrew installation command run until it has completed.

The Homebrew files are installed in the folder `/usr/local`. We now need to adjust the PATH variable so that the commands installed by Homebrew can be found. We do this by setting this variable in the `.bash_profile` file in your home directory with the nano editor. From the terminal, type:

```
mymac:~ me$ nano ~/.bash_profile
```

And write the following in that file:

```
mymac:~ me$ export PATH=/usr/local/bin:$PATH
```

Now save the changes with Ctrl-O (i.e., pressing both the control key and the O), and exit nano with Ctrl-X.

If you log out and log in again, the changes in the PATH variable should be effective. However, we can speed up this process, and test the edits in the `.bash_profile` file along the way, by entering:

```
mymac:~ me$ source ~/.bash_profile
```

Now install python3, with the command:

```
mymac:~ me$ brew install python3
```

You should see Homebrew giving you feedback about the installation process. This also installs the command pip3, with which we can install the additional python components.

Part 2: python components
We will install the python components in a *virtual environment*. This keeps the python installation separate from the Homebrew installation. From the terminal again, create the virtual environment with the command:

```
mymac:~ me$ python3 -m venv aepython
```

This will create a folder called aepython in your home folder. Before you can work with that environment, you need to activate it; enter the command:

```
mymac:~ me$ source aepython/bin/activate
```

You will see that the prompt of the terminal changes to something like:

```
(aepython) mymac:~ me$
```

That means you are using this newly created environment. Now install the needed parts, ypu can also type all moduel names after the first pip install, ju:

```
(aepython) mymac:~ me$ pip install numpy

(aepython) mymac:~ me$ pip install scipy

(aepython) mymac:~ me$ pip install matplotlib

(aepython) mymac:~ me$ pip install pygame

(aepython) mymac:~ me$ pip install spyder
```

Part 3: Using it
If you want to run python programs or use the editors, you need to enter the environment first, to do so open a terminal window and enter

```
mymac:~ me$ source aepython/bin/activate
```

To use idle (the default editor), then enter

```
(aepython) mymac:~ me$ idle
```

To use spyder (a more advanced editor), enter

```
(aepython) mymac:~ me$ spyder
```

If you have a python program and want to run it directly from the terminal, enter

```
(aepython) mymac:~ me$ python someprogram.py
```

Part 4 for later: Next year's courses
In the second year you will be using python packages for control theory (autopilot tuning and the like). To add these to your installation, you can use the following commands:

```
mymac:~ me$ brew install gcc

mymac:~ me$ source aepython/bin/activate
```

```
(aepython) mymac:~ me$ pip install slycot

(aepython) mymac:~ me$ pip install control

(aepython) mymac:~ me$ pip install PyDSTool
```

Note that the slycot installation step takes quite some time, because pip needs to compile the slycot Fortran code on your computer.

See also: https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-local-programming-environment-on-macos

1.3.4 Explanation of the installed modules

The description above installs a number of modules:

- **Scipy** will also install **Numpy**, giving you many tools for scientific calculations, also with large table or matrices.
- **Matplotlib** is a very useful plotter, which contains an interactive plot window from which you can save image to be used in your publications
- **Pygame** is a 2D graphics library, which allows you to make real-time animations using both shapes and image file, read individual keys from the keyboard, process mouse input, generate sound and play music files. It is compatible with Numpy, and many other packages. It is easy to use and uses graphics acceleration hardware if available, but will also work without.
- **Spyder** is a working environment mainly to used for interactively doing scientific computing (iPython)

If you ever want to install a package which is not found by pip, it might be useful to download the .whl (wheel file) which is used by pip, yourself from this site:

https://www.lfd.uci.edu/~gohlke/pythonlibs/

Download the correct wheel file yourself, open a Command prompt, go to the downloads folder (with cd command) and type "pip install" in the folder with this file followed by the filename, for example:

`pip install` *example-py36-x64-filename.whl*

1.3.5 Configuring and using Python

First: Change Working Folder to My Documents\Python
In Windows, IDLE will start in the Python program directory (folder) and this will therefore also be your default working directory. This is dangerous because you may overwrite parts of Python when you save your own programs. Therefore make a shortcut on your Desktop in which we

change the working folder to a more appropriate one. Right-click in the Start Menu on IDLE, select Copy and then Paste it on to your Desktop. Then right click Properties of this Shortcut and **change the working directory to the folder where you want to keep your Python code** (e.g. My Documents\Python).

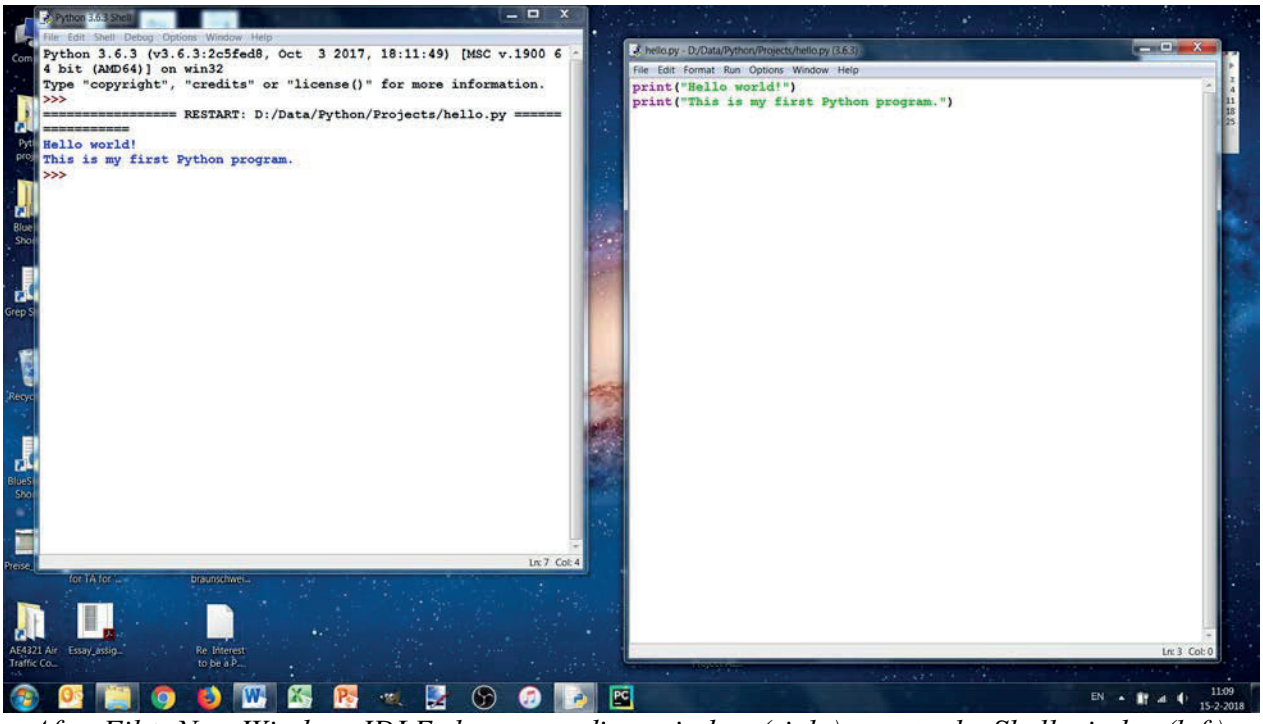
Add links to your documentation of the Help menu
Start IDLE. Then if you have not already done so, select in the menu of the IDLE window, **Options>Configure IDLE > General** to add additional Help sources in the lower part of the property sheet. Download the documentation for Scipy (CHM files) and use the link to the pygame/docs site as well as the Matplotlib gallery.

Using IDLE: your first program
In the IDLE window, named Python Shell, select **File > New Window** to create a window in which you can edit your first program. You will see that this window has a different pull-down menu. It contains "Run", indicating this is a Program window, still called Untitled at first. Enter the following lines in this window:

```
print("Hello World!")
print("This is my first Python program.")
```

Note: Python is case sensitive, so it matters that you use lower case for the print command.

Now select **Run > Run Module**. You will get a dialog box telling you that you have to Save it and then asking whether it is Ok to Save this, so you click Ok. Then you enter a name for your program like hello.py and save the file. The extension .py is important for Python, the name is only important for you. Then you will see the text being printed by the program which runs in the Python Shell window.



After File>New Window, IDLE shows an editor window (right) next to the Shell window(left)

Switching off the annoying dialog box "Ok to Save?"
By default, IDLE will ask confirmation for Saving the file every time you run it. To have this dialog box only the first time, goto Options>Configure IDLE>General and Select "No Prompt" in the line: **At Start of Run (F5)**. Now, on a Windows PC, you can run your programs by pressing the function key F5. Now only the first time you run your program, it will prompt you for a locations and filename to save it, the next time it will use the same name automatically.

1.3.5 Working environments: IDLE, PyCharm and Spyder

A working environment, in which you edit and run a program is called an IDE, which stands for Integrated Development Environment. Which one you use, is very much a matter of taste. In the course we will use as an editor and working environment the IDLE program, because of its simplicity. This is provided with Python and it is easy to use for beginners and advanced programmers. Since it comes with Python, it is hence also available in both distributions.

For larger projects or more advanced debugging, Spyder is recommended. Spyder is available in the Anaconda distribution and via pip. Spyder is mainly aimed at scientific Python an use a shell called iPython, with has some minor differences and is less suitable for programs that have graphics using pygame or OpenGL.

For more larger projects, there are many other IDEs, for example PyCharm. You try any of these. At the exam IDLE will be sufficient and Spyder is also available. So it is advised to learn how to use these two.

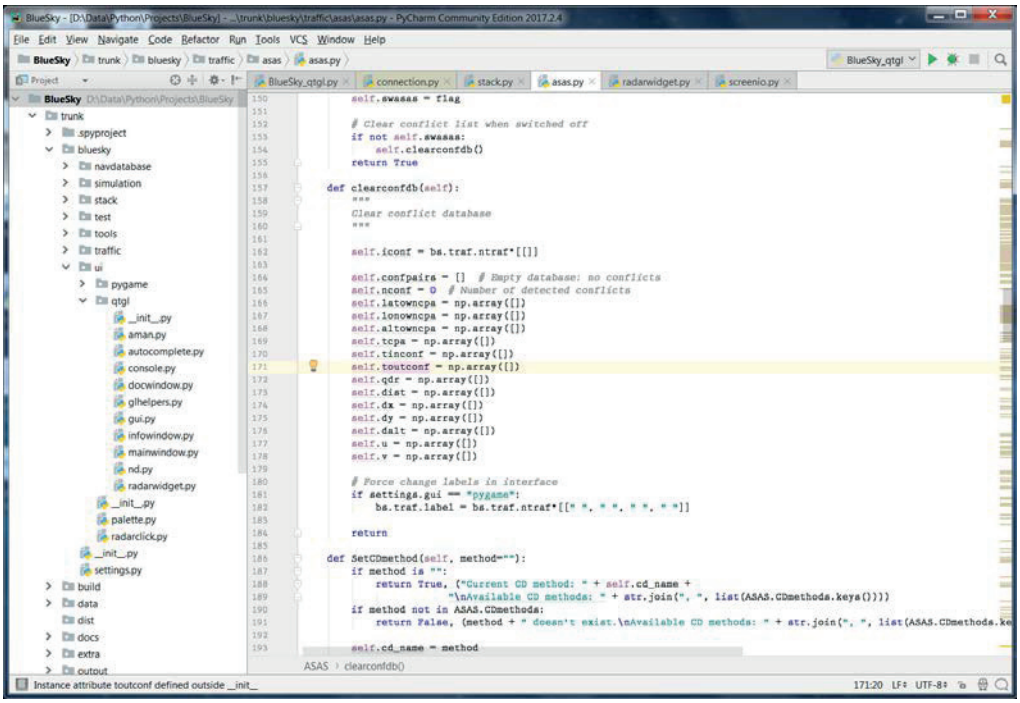
Though IDLE is a very useful IDE (Interactive Development Environment), there are some limitations:

- With large projects and many files it can become cumbersome to switch between different files
- Debugging facilities are limited

For this reason often another IDE is used for larger projects. There are many on the web.

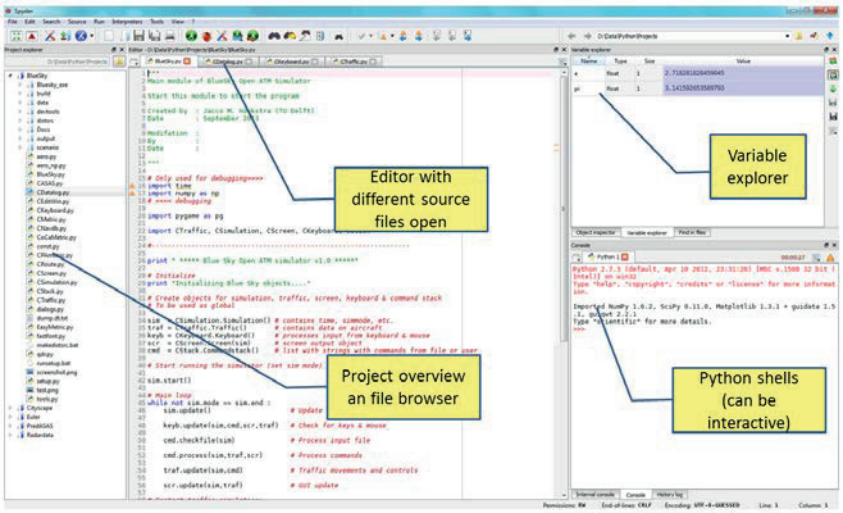
The screenshot on the next page shows PyCharm, a very popular versatile IDE. The community version, which we use, is freely available from Jetbrains:
https://www.jetbrains.com/pycharm/download/

Comparable options are Sublime and Atoms.



PyCharm IDE

For scientific purposes a very popular one is Spyder, as it reminds many older users of the Matlab tool for scientific computing which they used before. An example of a screenshot of Spyder with some explanation is given below:



Spyder screenshot

Other features include inspecting data arrays, plotting them and many other advanced debugging tools.

Make sure to change the settings of file in the dialog box which will pop up the first time you run the file to allow interaction with the Shell. Then you have similar features to which IDLE allows: checking your variables in the shell after running your program or simply to testing a few lines of code.

Spyder has chosen to stop offering a standard Python console, so they only have an iPython console. By tweaking the run setting per file, it is still possible to run your script in an external Python console. This would be a reason to prefer PyCharm as it is more versatile and focused on Standard Python applications not just the interactive iPython.

My advice would be to first keep it simple and use IDLE for the basics. Use the print( ) function and the shell (to inspect variables) as debugger and occasionally www.pythontutor.com. Then later, and only for larger or more complex problems switch to PyCharm (or optionally Spyder).

On the exam, only IDLE and SPyder will be available.

There are even more IDEs, here are some you could try out:

- Atom (open source shareware, multi-platform)
- Sublime
- Eclipse

A different environment especially for Scientific Computing, using iPython is Jupyter, which creates Python Notebooks, a beautiful blend between a document, spreadsheet and Python.

1.3.6 Documentation

IDLE, our default editor supplied with Python, has an option to Configure IDLE and add items to the Help menu. Here a link to a file or a URL can be added as an item in the Help pull down menu.

The Python language documentation is already included.

For Scipy and Numpy, downloading the .CHM files ('chum-files') of the reference guides onto your hard disk and linking to these files is recommended. They are available for download at:

http://docs.scipy.org/doc/

For Matplotlib both an online manual as well as a pdf is available at:

http://matplotlib.sourceforge.net/contents.html

Also check out the Gallery for examples but most important: with the accompanying source code for plotting with Matplotlib:

http://matplotlib.sourceforge.net/gallery.html

For Pygame, use the online documentation, with the URL:

http://www.pygame.org/docs/

Another useful help option is entering 'python' in the Google search window followed by what you want to do. Since there is a large Python user community, you will easily find answers to your questions as well as example source codes.

1.4 Sneak preview: Try the Python language yourself

1.4.1 Temperature conversion: (PRINT, INPUT statement, variables)

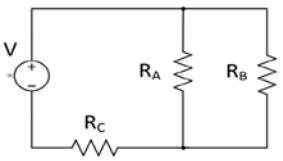
In the IDLE window, named Python Shell, select **File > New Window** to create a window in which you can edit your program. You will see that this window has a different pull-down menu. It contains "Run", indicating this is a Program window, still called Untitled at first.

Enter the following lines in this window and follow the example literally.

```
print("Temperature conversion tool")

tempf = float(input("Enter temperature in degrees Fahrenheit:"))
tempc = (tempf-32.0)*5.0/9.0

print(tempf,"degrees Fahrenheit is about",round(tempc),"degrees Celsius")
```

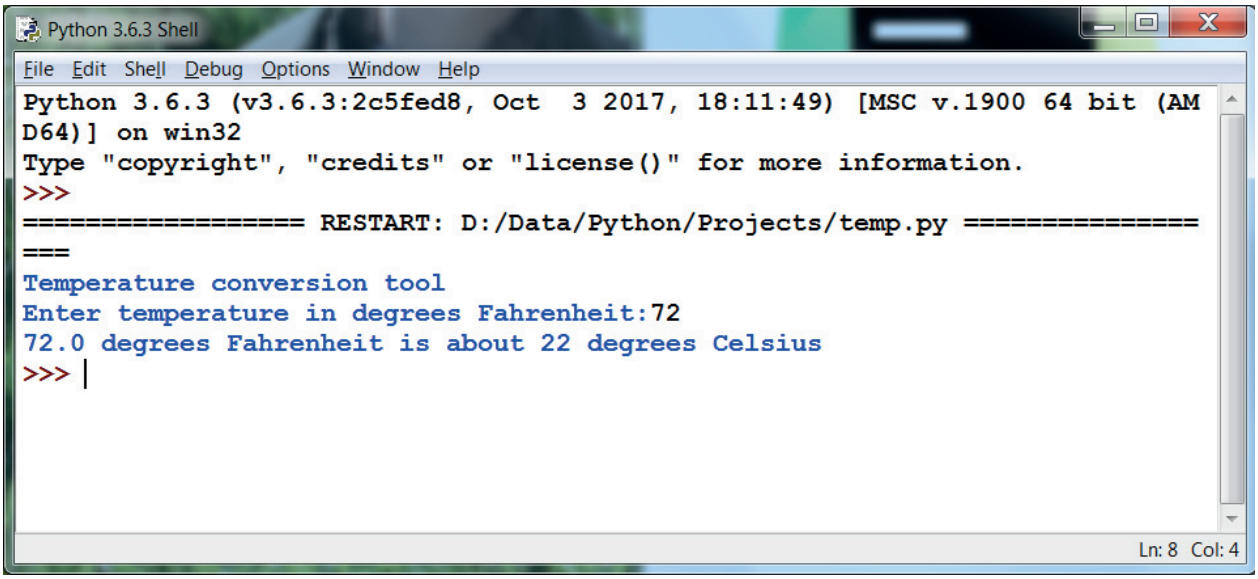
*Exercise 1: If you want to make it a bit more interesting, and harder for yourself, you could make a variation on this program. In much the same fashion, you could try to make a saving/debt interest calculator where you enter start amount, interest rate in percentage and number of years. To raise x to the power y, you use x**y)*

Exercise 2: Make a program that computes the overall resistance for the circuit shown at the right, where your inputs are the three resistances (R_A, R_B, R_C).



(Solutions to exercises can be found in Appendix D)

Now select **Run > Run Module**. Depending on your settings, you might get a dialog box telling you that you have to Save it and then asking whether it is Ok to Save this, so you click Ok. Then you enter a name for your program like temperature.py and save the file. The extension .py is important for Python, the name is only important for you. Then you will see the text being printed by the program, which runs in the window named "Python Shell":



Now let us have a closer look at this program. It is important to remember that the computer will step through the program line by line. So the first line says:

```
print("Temperature conversion tool")
```

The computer sees a print( ) function, which means it will have to output anything that is entered between the brackets, separated by commas,) to the screen. In this case it is a string of characters, marked by a " at the beginning and the end. Such a string of characters is called a **text string** or just **string**. So it put this on the screen. The computer will also automatically add a newline character to jump to the next line for any next print statement (unless you specify otherwise, which will be explained later).

Then this line is done, so we can go to the next one, which is slightly more complicated:

```
tempf = float(input("Enter temperature in degrees Fahrenheit:"))
```

This first part of this line, until the "=",should be read as: "let **`tempf`** be". This is a so called assignment statement, indicated by the "=" symbol. In general, it has the following structure:

variablename = ***expression***

In our example it tells the computer that in the computer's memory a variable has to be created with the name **`tempf`**. (If the variable name is already in use, it will be overwritten by this result at the end.)

To be able to do this, the computer first evaluates the expression on the other side of the "=" sign to see what the type of this variable has to be. It could for example be a floating point value (***float*** type) or a round number (***integer*** type), a series of characters (***string***) or a switch (***boolean*** or logical). It then reserves the required amount of bytes, stores the type and the name. If the name already exists, then this old value and type are first to avoid problems later on.

The computer evaluates the expression. The outcome is stored in memory and can be used later in other expressions by using the variable name on the left side of the equal sign. To do this, the computer saves the name in a table associated with this module. This table points to a memory address, helping to find the computer where the contents and type are stored in the computers memory.

```
a = 2
```

For numbers three are two basic types of variables:

- integers
- floats
- complex

Integers are whole numbers, used to count something or as an index in a table or grid. Floats are numbers with a floating point and can have any real value. Complex numbers refer to the mathematical definition of 'complex'. They are like floats but can have an imaginary part next to the real part. They are stored in a different way.

Python looks at the expression to determine the type:

```
 2  => integer type
-4  => integer type
3*4  => integer type

2.0  => float type
0.   => float type
1e6  => float type
3*4.  => float type

3+4j  => complex type
2+0j  => complex type
1j    => complex type
```

In chapter 2 the assignment statement, as well as the different types ofvariables are discussed in detail.

Now let us have a look at the expression in our example program. This is not a simple one. The expression in our example has a combination of twice the following structure :

functionname (argument)

Python knows this is a function because of the brackets. In our example case, there are two functions **float( )** and **`input()`.** As the result of the input-function is needed as input of the float-function, the input function is called first. Both are standard functions included in the Python language. (Later we will also use functions which we have defined ourselves!)

Most functions do some calculations and yield a value. Example of these functions are **`abs(x)`** for the absolute value (modulus) of x or **`int(x)`** which will truncates the float x to returns an integer type. The int( ) function is one of the type conversion functions:

`int(3.4)` => integer with value 3
`int(-4.315)` => integer with value -4
`float(2)` => float with value 2.0
`float(0)` => float with value 0.0

But some functions are complete little programs in itself. The input-functions for example does more: it can be called with one argument, which will be printed on the screen, before the user is prompted to enter a value. When the user presses enter, the value is read, the type is determined and this is returned as the result by the input function. So in our case, the text `Enter temperature in degrees Fahrenheit:` is printed and the user enters something (hopefully a number) and this is then stored as an integer or floating point number in a memory location. We call this variable tempf.

The next line is again an assignment statement as the computer sees from the equal sign "=":

```
tempc = (tempf-32.0)*5.0/9.0
```

Here a variable with the name `tempc` is created. The value is deduced from the result of the expression. Because the numbers in the expression on the left side of the equal sign are spelled like "`5.0`" and "`32.0`", the computer sees we have to use **floating point** calculations. We could also have left out the zero as long as we use the decimal point, so **`5./9.`** would have been sufficient to indicate that we want to use floating point values.

If we would leave them out, so using **`5/9`** would strictly speaking refer to dividing two integers. In Python 2 the result will hence be zero. However, in Python 3, this is converted to a float, as it often led to errors. Integer division is still available, but requires two slashes: **`5//9`** will yield zero as result in **integer arithmetic**. (See sections 2.4 and 2.5 for a detailed discussion of the difference between integers and floats).

When this expression has been evaluated, a variable of the right type (float) has been created and named **`tempc`**, the computer can continue with the next line:

```
print(tempf,"degrees Fahrenheit is",round(tempc),"degrees Celsius")
```

This line prints four things: a variable value, a text string, an expression which needs to be evaluated and another text string, which are all printed on the same line as with each comma a

space character is automatically inserted as well. The round function means the result will be rounded off.

Try running the program a few times. See what happens if you enter your name instead of a value.

1.4.2 Example: a,b,c formula solver (IF statement, Math functions)

Now create a new window and enter the program below

```
import math

print("To solve ax2 + bx + c = 0   :")

a = float(input("Enter the value of a:"))
b = float(input("Enter the value of b:"))
c = float(input("Enter the value of c:"))

D = b**2 - 4.*a*c

x1 = (-b - math.sqrt(D)) / (2.*a)
x2 = (-b + math.sqrt(D)) / (2.*a)

print("x1 =",x1)
print("x2 =",x2)
```

Run this program and you will see the effect. Some notes about this program:

- note how ** is used to indicate the power function. So 5**2 will yield 25. (Using 5*5 is faster by the way.)

- the program uses a function called sqrt() This is the square root function. This function is not a standard Python function. It is part of the math module, supplied with Python. Therefore the math module needs to be imported at the beginning of the program. The text math.sqrt() tells Python that the sqrt() function can be found in the imported math module

- After you have run the program, you can type D in the shell to see the value of the discriminant. All variables can be checked this way.

Also, note the difference between text input and output. There is no resulting value returned by the **`print`** function, while the **`input`** function does return a value, which is then stored in a variable. The argument of the input-function calls is between the brackets: it's a prompt text, which will be shown to the user before he enters his input.

There is one problem with our program. Many times it will stop with an error because the discriminant D is negative, resulting in an error with the square root function.

To solve this, let us try adding some logic to the program, see below. Adapt your program to match this precisely, note the margin jumps (use TAB-key) in the IF statement, which is called **indentation**.

```
from math import sqrt

print("To solve ax2 + bx + c = 0")

a = float(input("Enter the value of a:"))
b = float(input("Enter the value of b:"))
c = float(input("Enter the value of c:"))

D = b**2 - 4.*a*c

if D<0:
    print("This equation has no solutions.")

else:
    x1 = (-b - sqrt(D)) / (2.*a)
    x2 = (-b + sqrt(D)) / (2.*a)

    print("x1 =",x1)
    print("x2 =",x2)
```

Now the program first checks whether D is negative. If so, it will tell you that there are no solutions.

- Note the structure and syntax(=grammar) of the if-else statement. A **colon** (:) indicates a block of code will be given, so it acts as a 'then'. The block is marked by the **indented** part of the code,.

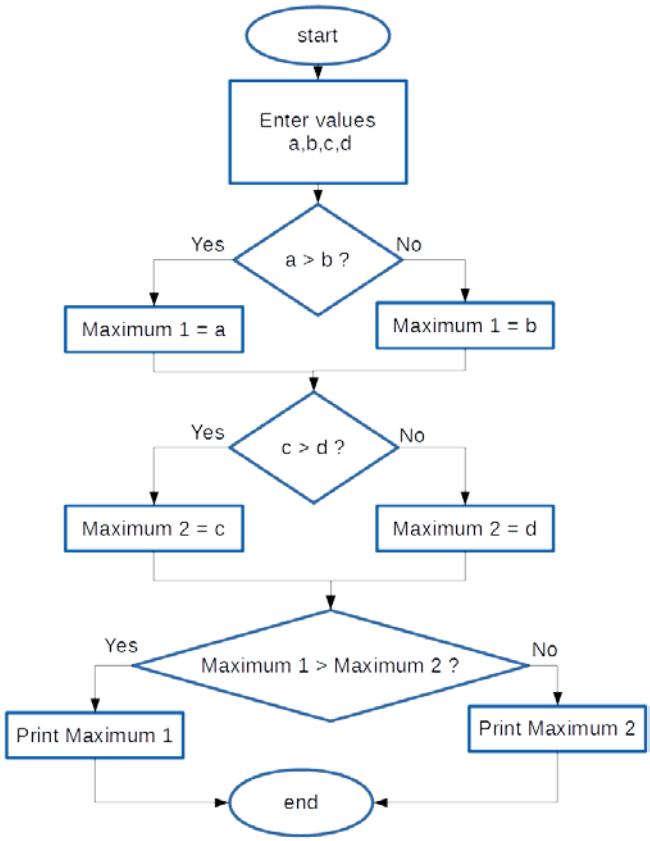
- When it jumps back to the margin of the beginning to indicate the end of this block of code, an '**else**' follows, again with a colon and an indented block of code.

- It also uses a different way to **import** the **sqrt function** from the **math module**. Note the difference in syntax in both the line with the import statement as well as the line where the sqrt function is used.

Exercise 1.1:
Adapt the program so that it calculates the hypotenuse c of a rectangular triangle for rectangular sides a and b as entered by the user, using Pythagoras formula.

Exercise 1.2:
See the flow chart on the right.

Using this flow chart, make a program that determines the maximum value of given numbers a ,b, c and d as entered by the user, using a number of if-else-statements to find the maximum value of these four variables.



1.4.3 Example: using lists and a for-loop

Now let us have a look at a program which is slightly more complex. First explore the range function. Go to the Python shell and type the following lines to see how the range-function works.

```
list(range(10))

list(range(1,11))

list(range(2,22,2))

list(range(5,1,-1))
```

What do you notice? If you do not see its logic, try a few values yourself. Some things you probably have noticed:

- The list function converts the range function's result to a list of integers separated by a comma in between square brackets:
  [2, 3, 4, 5]
- the range-function has three arguments start, stop and step The stop is always required, but start en step are optional
- the default start value is zero
- the default step value is one
- the start value is included in the list
- the stop value is not included in the list

This **list** is in fact a new variable type:**.** You can regard this as a table:

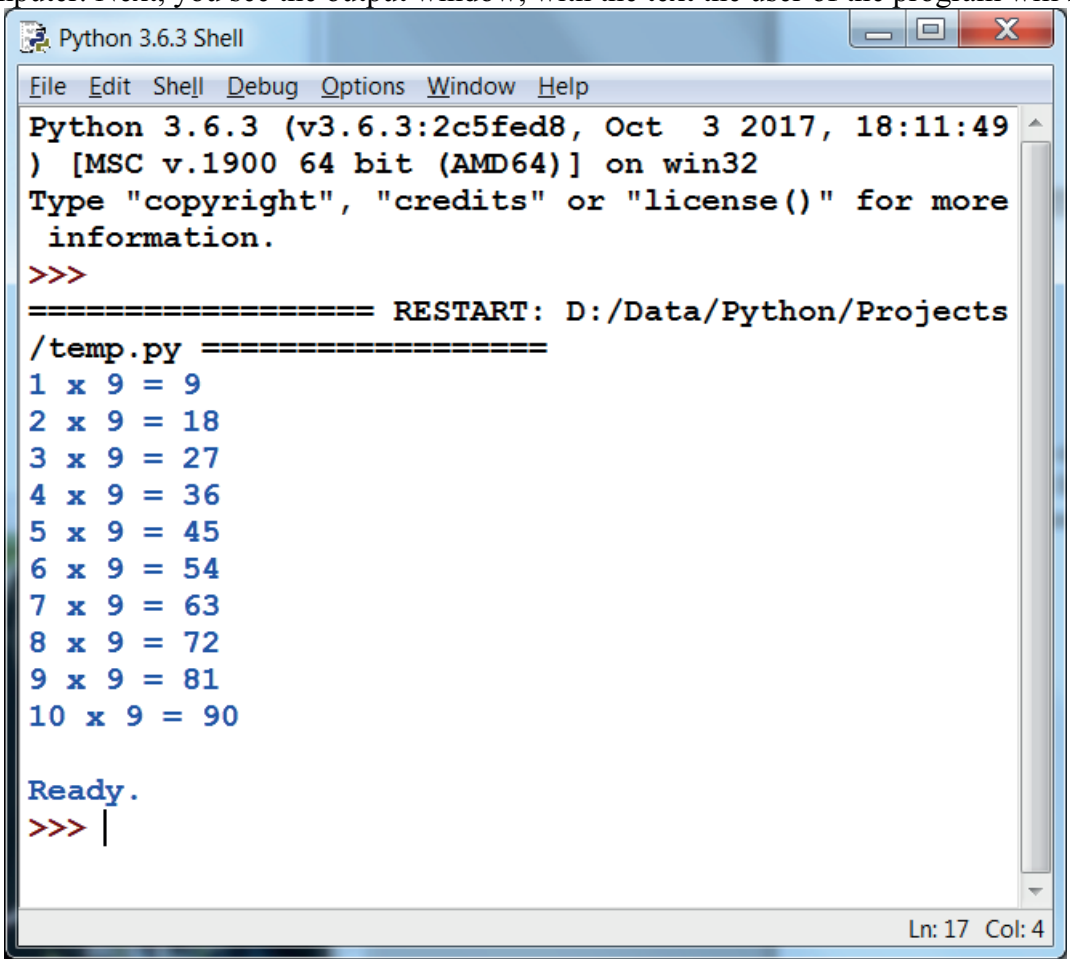
```
a = [ 7, 3, -1, 3]
```

Indexing of the list starts with zero, so a[0] will return in the first value (7) and a[3] the last one (3).

For the next example code we will use the website http://www.pythontutor.com . Go to this site and on the start page click "Start using..", clear the source edit window and enter this program in the window (also mind the layout, use tab to move the margin right!)

```
a = 9

for i in range(1,11):

    x = i*a
    print (i,"x", a,"=",x)

print()
print("Ready.")
```

Now click “Visualize execution” on the website and then click “Forward” a few times to see what happens. On the right side of the edit window you can see what happens in the memory of the computer. Next, you see the output window, with the text the user of the program will see:



Can you explain what the computer does? Why does he jump back? How does he know which part of the code to repeat and which not? Do you notice what happens to the value of i when it jumps back to the for-statement? What happens when i has reached the end of the sequence?

This is called a for-loop: i is assigned to the first value of the list (in this case the list made by the range function) and after it has completed the indented block of code, it jumps back and assigns the next value of the list until it has reached the end of the list. If there are no more values for i, so after the last value, it continues the code and does not jump back, the variable i now has the final value (10, because 11 is not included in the list generated by the range-function). See also the program below, what will this program do? Which integer do you think the len( ) function returns?

```
lst = [ 40. ,  5. ,  13.,  1.,  5. ]
for i in range(len(lst)):
    print(2*lst[i])
```

Notice the difference in syntax between calling a function in Python:

```
sqrt(D)
len(a)
range(1,11)
```

and the use of a list with indices:

```
a[0]        # to get the first element use index zero!
lst[i]      # when i=1, you get the second element, etc.
```

You can see that Python knows whether something is a list or a function based on the type of brackets used!

See below an example of a table lookup using a for-list:

```
# Enthalpy of water at 1 atmosphere

Ttab = [0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
Htab = [0.06,42.1, 84.0, 125.8,167.5,209.3,251.2,293.0, 335.0, 377.0,419.1]

n = len(Ttab) # Length of list

T = float(input("Enter temperature in degrees Celsius:"))

for i in range(n-1):
    if T >= Ttab[i] and T < Ttab[i+1]:
        print("H is between",Htab[i],"and",Htab[i+1])
```

Note the two indentations: one for the for-loop, the next for the if statement.

A unique feature of Python is that the same list can store different types of variables:

```
b = [2, "Hello there!", 3.141565 , 2, 10.0, True]
```

This assignment of b is a valid list, and it consists of a mix of variable types: floats, integers, a string and a Boolean (logical)

You even store lists in a list:

```
c = [  [2, 3, -1] , [3, 4, 0] , [7, 1, 1] ]
```

And the result is basically a two dimensional table, as we can see by showing some values of this table in the Python shell (first type the assignment statement above):

```
>>>c[0]

[ 2, 3, -1 ]

>>>c[2][0]

7
```

The second c[2][0] basically means, from left to right: the third element of c (which is a list) and then the first element of that list.

With this format it is easy to select a row, but selecting a column is only possible with a for loop. Below there are two ways to go through a two dimensional list to pick a column, What would be advantages of each method?

The first method is to have an integer run through a list of integers (so whole numbers) as generated by the range function: [0,1,…..len(people)-1] Remember the end value given in the range function will not be included in the range functions resulting list. These are exactly the indices for the list as this also starts with 0 and ends with its length minus one.

```
# Database: one statement can cover more program lines
names = ["Jan","Piet","Kees","Klaas","Victor"]
age   = [18,20,19,34,22]
city  = ["Delft","Leiden","Amsterdam","Utrecht",       \
           "Leeuwarden"] ]

# Show first two lists
for i in range(len(names)):
    print(names[i],"is",age[i],"years old.")
```

The above way will work in most other programming languages as well. A unique feature of python is that a list of any type can be used as the counter (or as we call it: iterator) in the loop. The variable person will get each value from the list people. As people is a list of lists, person will first be the first element from people: `["Jan", 18, "Delft".` Then, when the block of code that is in the loop has been executed with this value for person, the next value of people will be used: `["Piet", 20, "Leiden"]` and so on, for as long as the list people lasts:

```
# Database: see how one statement can cover more program lines
# (within a list definition, without the backslash is also ok)

people = [   ["Jan",  18, "Delft"],         \
             ["Piet", 20, "Leiden"],        \
             ["Kees", 19, "Amsterdam"],     \
             ["Klaas",34, "Utrecht"],       \
             ["Victor",22,"Leeuwarden"] ]

iname = 0
iage  = 1
icity = 2

# Show first two columns of table
for person in people:
    print(person[iname],"is",person[iage],"years old.")
```

Another way to store this database is to use a so-called dictionary, using the other brackets { } . Then elements from this dictionary can be used using not numbers as an index in the square bracket, but a so-called *key*. See the example below:

```
# Using dictionaries {key:value,key:value, etc. }
```

```
age  = {"Jan":18 , "Piet":20, "Kees": 19, \
        "Klaas":34, "Victor":22 }

city = {"Jan" : "Delft", "Piet" : "Leiden",              \
        "Kees" : "Amsterdam","Klaas" : "Utrecht", \
        "Victor" : "Leeuwarden"}

# Show names and ages

for person in age:
    print(person,"is", age[person],"years old.")
```

Lists are the basic type. Lists are often created by appending values at the end of the list, using the append function, which comes with the list-type and has a special syntax (varname.function), similar to how we use functions from a module, which we will later see more often. Such a function, which is called by a dot after the variable name is called a method, in this case of the list object (i.e. the list type).

Try this bit of code:

```
debt = []
rate = 1.03
x = 30000.
for i in range(30):
      debt.append(x)
      x = x*rate
print(debt)
```

Now try to change the line with the append function in:

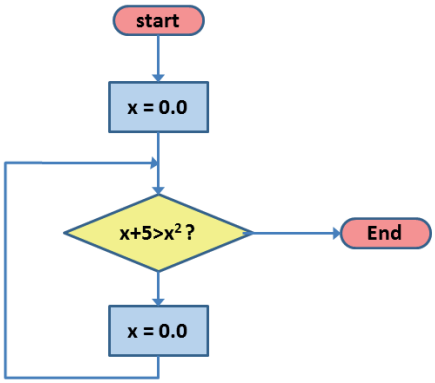
```
debt.append([i,x])
```

and see what the effect is (and what the debt is after 30 years of only 3% interest!). Could you think of a way to make the output look better, using a for-loop, list-indices, the e.g. the round( ) function (see section 2.8)?

1.4.4 While loops example

The final basic statement which will complete your basic Python vocabulary is the while statement. It has been formally proven that with IF and WHILE you can program any logic you can think of. The FOR loop is basically a special case of the while-loop, for convenience. So how does the while loop work? It is basically an IF statement which will repeat the indented block of code until the condition becomes false. See the example below:

```
x = 0.

while x+5>x*x:
    print(x)
    x = x+1
```



What do you think the output of this program will be?
The flow chart of this program would look like this:

Also notice, that input always creates a text variable, a so-called string, which still needs to be converted to the right type.

Or see how this while-loop finds the right spot in a table to interpolate:

```
# Enthalpy of water at 1 atmosphere

Ttab = [0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
Htab = [0.06,42.1, 84.0, 125.8,167.5,209.3,251.2,293.0, 335.0, 377.0,419.1]

n = len(Ttab) # Length of list

T = float(input("Enter temperature in degrees Celsius:"))

i = 0
while T>Ttab[i+1] and i+1<n-1:
    i = i + 1

print("H is between",Htab[i],"and",Htab[i+1])

f   = (T-Ttab[i])/(Ttab[i+1]-Ttab[i])
Hip = (1.-f)*Htab[i] + f*Htab[i+1]

print("My best guess is that H will be:",Hip)
```

In the following example a while loop is used to find a value in a list:

```
# List of brands and products
brands = ["Apple","Samsung","Airbus","Renault","Microsoft","Google"]

products = [["computers","tablets","cell phones","music"],
            ["Electronics"],
            ["Airliners","Transport aircraft"],
            ["Cars"],
            ["Software"],
            ["Search engines","Operating Systems"]]

# String input function
br = input("Give a brand name:")

# Initialize loop parameters: counter and boolean/logical variable
i = 0
found = False

# Can you see why we use the len(brands)-1 here?
while i<len(brands)-1 and not found:
    if not brands[i] == br:
        i = i + 1
    else:
        found = True

# Show result of search loop
if found:
    print("They make:",products[i])
else:
    print("I don't know the brand",br,"!")
```

1.4.5 More modules

Python comes with many handy features built-in as modules. To be able to access these from your program, simply put an import statement at the beginning. Then in your program simply type the module name followed by a period and the function name. In this way you can access all functions inside this module. Some examples are given in this section.

Type in the shell `help("time")` or `help("time.localtime")` and try to see what happens in the program below. It uses the time module to get current local time and date as integers (whole numbers).

```
import time

# Get local time & date
t = time.localtime()

# Using the tuple(= list with fixed values)

hour = t[3]
mins = t[4]
secs = t[5]

date  = t[2]
month = t[1]
year  = t[0]

# Or use the time structure . See help("time.time")
hour = t.tm_hour
mins = t.tm_min
secs = t.tm_sec

date  = t.tm_mday
month = t.tm_mon
year  = t.tm_year
```

Or the random number generator from the moduel named `random` (in the shell type `help("random.random")` and `help("random.randint")` to get more information:

```
import random

# Two ways to get a number 1-6
die = int(random.random()*6)+1
print(die)

die = random.randint(1,6)
print(die)
```

Exercise 1: How would you simulate a coin toss with the random method?

Exercise 2: How would you simulate the picking of a random playing card?

Exercise 3: How many seconds there are left in 'today' till tomorrow?

1.4.6 Finding your way around: many ways in which you can get help

Using `help("text")` or interactive `help()`
If you wonder how you could ever find all these Python-functions and function modules, here is how to do that.

There are many ways to get help. For instance if you need help on the range function, in the Python shell, you can type:

```
help("range")
```

Which calls the help-function and uses the string to find appropriate help-information. Similarly to find methods of the list or string type, use:

```
help("list")
```

You can also use help interactively by typing help(), without arguments, and then type the keywords at the "help>" prompt to get help, e.g. to see which modules are currently installed.

```
>>>help()
help>math
```

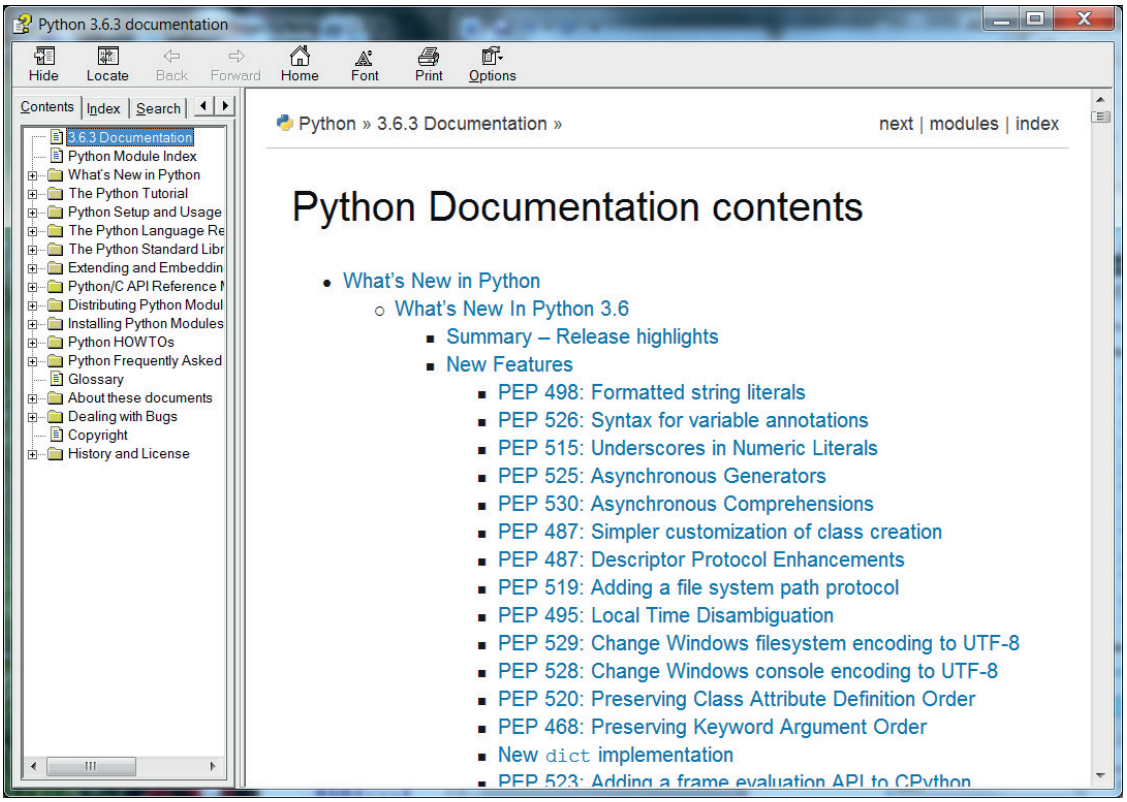
And you will see an overview of all methods in the math module. There are some limitations to this help. When you will type append, you will not get any results because this is not a separate function but a part of the list object, so you should have typed

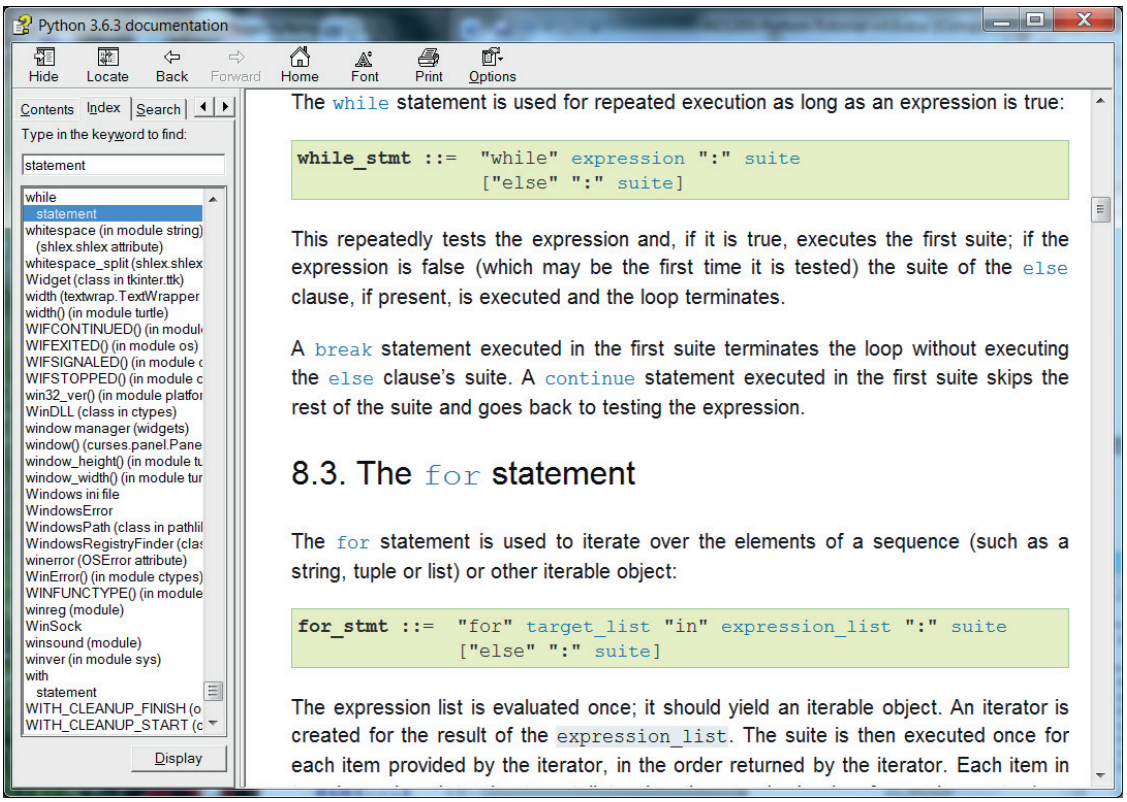
```
>>> help("list.append")
Help on method_descriptor in list:

list.append = append(...)
    L.append(object) -- append object to end

>>>
```

or list.append in the interactive help:

```
>>> help()
Welcome to Python 3.6's help utility.
…..
help> list.append
Help on method_descriptor in list:

list.append = append(...)
    L.append(object) -- append object to end

help>
```

Python documentation in Help Pull-down menu
So anther way is to use the supplied CHM-file, (compiled HMTL) via the Help-menu of the IDLE-windows: Select Help>Python Docs and you will find a good set of documentation, which you search in the "Index" or do a full text search ("Search"), see the screenshots on the next page:


Python 3.6.3 documentation
Hide
Locate
Back
Forward
Home
Font
Print
Options
Contents
Index
Search
3.6.3 Documentation
Python Module Index
What's New in Python
The Python Tutorial
Python Setup and Usage
The Python Language Re
The Python Standard Libr
Extending and Embeddin
Python/C API Reference
Distributing Python Modul
Installing Python Modules
Python HOWTOs
Python Frequently Asked
Glossary
About these documents
Dealing with Bugs
Copyright
History and License
Python » 3.6.3 Documentation »
next | modules | index
Python Documentation contents
What's New in Python
What's New In Python 3.6
Summary – Release highlights
New Features
PEP 498: Formatted string literals
PEP 526: Syntax for variable annotations
PEP 515: Underscores in Numeric Literals
PEP 525: Asynchronous Generators
PEP 530: Asynchronous Comprehensions
PEP 487: Simpler customization of class creation
PEP 487: Descriptor Protocol Enhancements
PEP 519: Adding a file system path protocol
PEP 495: Local Time Disambiguation
PEP 529: Change Windows filesystem encoding to UTF-8
PEP 528: Change Windows console encoding to UTF-8
PEP 520: Preserving Class Attribute Definition Order
PEP 468: Preserving Keyword Argument Order
New dict implementation


Python 3.6.3 documentation
Hide
Locate
Back
Forward
Home
Font
Print
Options
Contents
Index
Search
Type in the keyword to find:
statement
while
statement
whitespace (in module string)
(shlex.shlex attribute)
whitespace_split (shlex.shlex
Widget (class in tkinter.ttk)
width (textwrap.TextWrapper
width() (in module turtle)
WIFCONTINUED() (in modul
WIFEXITED() (in module os)
WIFSIGNALED() (in module
WIFSTOPPED() (in module
win32_ver() (in module platfor
WinDLL (class in ctypes)
window manager (widgets)
window() (curses.panel.Pane
window_height() (in module t
window_width() (in module tur
Windows ini file
WindowsError
WindowsPath (class in pathli
WindowsRegistryFinder (cla
winerror (OSError attribute)
WinError() (in module ctypes)
WINFUNCTYPE() (in module
winreg (module)
WinSock
winsound (module)
winver (in module sys)
with
statement
WITH_CLEANUP_FINISH (o
WITH_CLEANUP_START (c
Display
The while statement is used for repeated execution as long as an expression is true:
while_stmt ::= "while" expression ":" suite
["else" ":" suite]
This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates.
A break statement executed in the first suite terminates the loop without executing the else clause's suite. A continue statement executed in the first suite skips the rest of the suite and goes back to testing the expression.
8.3. The for statement
The for statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:
for_stmt ::= "for" target_list "in" expression_list ":" suite
["else" ":" suite]
The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the expression_list. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in

Using the huge Python community

Python has a very large user community: conservative estimates say there are 3 to 5 million Python users and it's growing fast as MIT, Stanford and many others use Python in their programming courses and exercises. It is also the most popular language among PhDs. IEEE calls it the standard language for scientific computing and data analysis.

So simply Googling a question (or error message) with the keyword Python (or later Numpy) in front of it as an extra search term will quickly bring you to a page with an example or an answer. For basic questions this might bring you to the same documentation as in the Help-menu which is at http://docs.python.org . You will also see that www.stackoverflow.com is a site, which will often pop-up, and where most questions you might have, have already been posed and answered:

For instance, for "python append to a list" you will find:
http://stackoverflow.com/questions/252703/python-append-vs-extend

(Which shows that next to **`append()`** there apparently is another method called **`extend()`** is available, which works with a list as an argument and apparently appends each element to the list.)

In general, thanks to the interactive Python Shell window, simply trying statements or program lines in the Python shell is a way to try what works as you intended. Or to check the value of variables after your still incomplete program has run.

To find bugs in your program, for small programs http://www.pythontutor.com can be helpful to see what is going on inside the computer when your program runs. And of course using the print statement to see what a variable is, or where the computer gets stuck in your program, is always the first thing to try.

Be prepared that in the beginning the level of frustration might be high, but so is the reward when your program runs. And when you get experience, you will see how fast you can make working programs, not because you won't create bugs, but because you will immediately recognize what went wrong. The nice thing of software is that it can always be fixed, so "trial and error" is an accepted way to develop programs. So do not be afraid to try things and play around with Python to get acquainted with the way it works.

In appendix A and appendix B a short overview of the Python statements and most important functions is given. Having a copy of these pages at hand may be handy when starting Python. It will provide the keywords for you to look into the help-functions of Google searches. Appendix C given some example source code.

Although we already touched upon most basic elements of the Python language, we have not seen all statements and types yet, so first go through the next chapters (discover the very useful while-loop, the string type and its methods and the Boolean/logical and slicing of strings and lists etc.). Just glance over these chapters to get the essence and then start making simple programs and study the paragraphs in more detail when you (think you) need them. Programming has a steep learning curve at the beginning, so good luck with making it through this first phase. Once you have passed this, things will be a lot easier and more rewarding. Good luck!

2. Python syntax: Variables and functions

In this chapter we describe the building blocks which you need to write a program. First we have a look at variables. A variable is used to store data. But there are different types of data: numbers or bits of text for instance. There are also different data types of numbers Of course there are more data types than just text and numbers, like switches (called “booleans” or “logicals”) and tables (“lists”). The type of variable is defined by the assignment statement, the programming line giving the variable its name, type and value. Therefore we first concentrate the assignment statement and then the different data types are discussed.

2.1 Assignment and implicit type declaration

A variable is a memory location with a name and a value. You define a variable when you assign a value or expression to it. This expression also determines the type. The assignment statement is very straightforward:

variablename = expression

Example assignments of the four main types (which will be discussed in details in the following sections):

Integers:

```
i = 200
a = -5
```

Floats:

```
in  = 2.54
ft  = .3048
lbs = 0.453592
spd = 4.
alt = 1e5
```

Logicals:

```
swfound  = True
alive    = False
forward  = (v > 0.0)
onground = h <= 0.0 and abs(v) < eps
running = (forward or alive) and not onground
```

Strings:

```
name = "Ian " + "McEwan"
txt = "I'm afraid I can't do that, Dave."
s   = 'HAL'
```

Python even allows multiple assignments at once when the values are separted by a comma:

```
a,b,c = 1. , 3. , 5
a,b = b,a                # swap two variables
```

What really happens is that the ezxpression first becomes a type of list (a tuple to be exact) which is then unpacked. The example below shows that this principle also works with a list:

```
lst   = [9 , 2 , 5]
a,b,c = lst
```

2.2 Short-hand: operator with equal sign

To increase the variable i with one, we normally type:

```
i = i + 1 # increase i with one
```

Mathematically speaking this statement would be nonesen if we would read the equal sign as 'being equal to'. But as an assignment statement, this simply evaluates the expression on the right side, which is the content fo variable i to which one is added, and the result is stored in the variable i again. The effect is thus that i is increased with i. Here are a few other example with the same logic:

```
j = j - 1  # decrease j with one
n = n*2    # doubles n
x = x/2    # halves x
a = a*k    # a will be multiplied by k
```

As especially the increase or decreases with one occurs often, there is a shorthand notation for things:

```
" i = i + "  => " i +="
```

So the examples above can also be coded, like this:

```
i += 1    # increase i with one
j -= 1    # decrease j with one
n *= 2    # doubles n
x /= 2    # halves x
a *= k    # a will be multiplied by k
```

This short-hand is the same as in the programming language C (and C++), a language which is well known for how short code can be written, but often with an adverse effect on the readability. Also in this example, the benefit is mainly for the programmer who saves a few keystrokes, but there is no execution time benefit as the operations done are the same. It does make your code less readable for non-programmers as it requires them to know that "i+=" means "i = i +", which is not obvious if you have never see this before. Part of the charm of Python is that you can write efficient code which still is understandable even by non-programers, so that the source is also function as documentation of the system. This short-hand hence decreases the readability.

2.3 Number operations in Python

For the number types float, integer and complex the following operators can be used

+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floored division (rounded off to lower round number)
%	Modulo, remainder of division
**	Power operator x**y is x to power y, so x**2 is x^2
-x	Negated value of x
+x	Value of x with sign unchanged

As well as the fillowing standard functions:

abs(x)	Absolute value of x , |x|
pow(x,y)	Power function, same as x**y
divmod(x,y)	Returns both x//y and x%y

2.4 Floats: floating point values

Floating point variables, commonly referred to as floats, are used for real numbers with decimals. These are the numbers as you know them from your calculator. Operations are similar to what you use on a calculator. For power you should use the asterisk twice, so x^y would be `x**y`.

They are stored in a binary exponential way with base 2 (binary). This means that values which are round values in a decimal systems sometimes need to be approximated. For insatcne if you try typing in the shell:

```
>>> 0.1+0.2
```

The result will be:

```
0.30000000000000004
```

Often this is not a problem, but it should be kept in mind, that float are always approximations!

One special rule with floats in programming is that you **never test for equality with floats**. So never use the condition “when x is equal to y” with floats, because a minute difference in how the float is stored can result an inaccuracy, making this condition False when you expected otherwise. This may not even be visible when you print it, but can still cause two variables to be different according to the computer while you expected them to be the same. For example: adding 0.1 several times and then checking whether it is equal to 10.0 might fail because the actual result is approximated by the computer as 9.99999999 when it passes the 10. So always test for smaller than (<) or larger than (>). You can also use “smaller or equal to” (<=) and “larger or equal to” (<=), but do not forget that a floating point variable is always an

approximation due to the binary storage limitations (which is different from a decimal notation), so it may be off by a small value. A way to avoid it is to test for closeness:

```
# Test for equality with floats

eps = 1e-7        #value depends on scale you're looking at
if abs(x-y)<eps:
    print("x is equal to y")
```

Floating point values can also be rounded off using the round(x) or round(x,n) function. With the second argument n, you can specify the amount of decimals. When the second argument isnot used, the result will be rounded off to the nearest value and converted to the type integer automatically.

```
>>> round(10/3)
3
>>> round(10/3,4)
3.3333
>>> round(10./3.,0)
3.0
```

Try adding 0.1 to a variable for more than 1000 times (use a for-loop or while-loop) and check the variable value to see this effect.

2.5 Integers

Integers are variables used for whole numbers. They are used for example as counters, loop variables and indices in lists and strings. They work similar to floats but **division** results will be of the type float. For example:

```
a = 4
b = 5
c = a/b
print(c)
```

This will print 0.8 (in Python2 it would always floor this to an integer, but in Python 3, this has been changed).

A very useful operator, especially with integers, is the ' %' operator. This is called the **modulo function**. You could also call it "the remainder" because it gives only remainder of a division. For example:

```
27%4 =  3
4%5  =  4
32%10 = 2
128%100 = 28
```

So to check whether a number is divisible by another, checking for a remainder of zero is sufficient:

```
if a%b==0:
      print("b is a factor of a")
```

You can convert integers to floats and vice versa (since they are both numbers) with the functions **int()** and **float()**:

```
a = float(i)
j = int(b)          # Conversion cuts of behind decimal point
k = int(round(x))   # Rounded off to nearest whole number
                    # ..and then converted to an integer type
```

The **int()** function will cut off anything behind the decimal point, so int(b) will give the largest integer not greater than b . These functions **int()** and **float()** can also take strings (variables containing text) as an argument, so you can also use them to convert text to a number like this:

```
txt = "100"
i = int(txt)
a = float(txt)
```

As in Python 3 a deviation from most other languages has been implemented, whenever a fraction results from a integer division, the result is then a float. This might also yield unexpected results, especially as Python 2 and other programming languages do not have this behaviour. To avoid this there are several ways:

```
n = 20
i = 3
a = n/i
print("a =",a)

j = n//i        # floor division
k = int(n/i)    # rounded off to below and converted to int
m = round(n/i) # rounded off to nearest integer value
print("j,k,m =",j,k,m)
```

The output of this program will be:

```
a = 6.666666666666667
j,k,m = 6 6 7
```

As we can see from the missing decimal points in the second line, the variable **j, k, m** all are integers. Whick can also be verified in the shell by using the **type( )** function:

```
>>> type(m)
<class 'int'>
```

2.6 Complex numbers

Python also has cpmplex numbers built-in as standard type. In many other languages one would have to treat this as two float numbers, the real part and the imaginary part, but in Python the j directly after a number signals an imaginary art and will change the resulting type to complex.

```
c = 3+4j
print ("Length of",c,"is",abs(c))
```

Will give the following output:

```
Length of (3+4j) is 5.0
```

To access the different components, use the following method:

```
c = 3+4j
a = c.real
b = c.imag
print("c has",a,"as real part and",b,"as imaginary part")
```

Will give the following output:

```
c has 3.0 as real part and 4.0 imaginary part
```

All other operations work as expected:

```
>>> e  = 2.718281828459045
>>> pi = 3.141592653589793
>>> e**(1j*pi)
(-1+1.2246467991473532e-16j)
```

Which demonstrates Euler's equation and also shows the rounding off problems in the imaginary part, which is practically zero. Both e and pi were approximations.

2.7 String operations

Strings are variables used to store text. They contain a string of characters (and often work similar to a list of characters). They are defined by a text surrounded by quotes. These quotes can be single quotes or double quotes, as long as you use the same quote-character to start and end the string. It also means you can use the other type of quote-symbol inside the text. A useful operator on strings is the "+" which glues them together, Which can be very useful, e.g in functions which expect one string variable (like the input function).
Example assignments:

```
txt = "abc"
s = ""
name = "Jacco"+" M. "+"Hoekstra"   (so the + concatenates strings)
words = 'def ghi'
```

```
a = input("Give value at row"+str(i))
```

Some useful basic functions for strings are:

len(txt)	returns the length of a string
str(a)	converts an integer or float to a string
eval(str)	evaluates the expression in the string and returns the number
chr(i)	converts ASCII code i to corresponding character (see table on next page)
ord(ch)	returns ASCII code of the character variable named ch

Using indices in square brackets [ ] allows you to take out parts of a string. This is called slicing. You cannot change a part of string but you can concatenate substrings to form a new string. This can be used to achieve the same thing:

c = "hello world"
c[0] is then "h" (indices start with zero)
c[1:4] is then "ell" (so when 4 is end, it means until and not including 4)
c[:4] is then "hell"
c[4:1:-1] is then "oll" so from index 4 until 1 with step -1
c[::2] is then "hlowrd"
c[-1] will return last character "d"
c[-2] will give one but last character "l"
c = c[:3]+c[-1] will change c into "hel"+ "d"="held"

the string variable also has functions built-in, the so-called string methods. See some examples below (more on this in chapter 8).

a = c.upper()	returns copy of the string but then in upper case
a = c.lower()	returns copy of the string but then in lower case
a = c.strip()	returns copy of the string with leading and trailing spaces removed
sw = c.isalpha()	returns True if all characters are alphabetic
sw = c.isdigit()	returns True if all characters are digits
i = c.index("b")	returns index for substring in this case “b”

For more methods see chapter 8 or 5.6.1 of the Python supplied reference documentation in the Help file.

Strings are stored as lists of characters. Characters are stored as numbers, e.g. using their ASCII codes. See the table below:

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

Special characters in a string are: \n (newline) or \t (tab)
In Windows/DOS text files the newline is indicated by characters: both a Carriage Return & Line Feed: chr(13)+chr(10). While in Linux/Apple it is only a newline character chr(10)

2.8 Logicals/Booleans

Logicals or Booleans are two names for the variable type in which we store conditions. You can see them as switches inside your program. Conditions can be either `True` or `False`, so these are the only two possible values of a logical. Mind the capitals at the beginning of `True` and `False` when you use these words: Python is case sensitive. Examples of assignments are given below:

```
found = False
prime = True
swlow = a<b
outside = a>=b
```

```
swfound = a==b
notfound = a!=b     ( != means: not equal to)
notfound = a<>b     ( <> also means: not equal to, but is old notation)
outside = x<xmin or x>xmax or y<ymin or y>ymax
inside = not outside
out = outside and (abs(vx)>vmax or abs(vy)>vmax)
inbetween = 6. < x <= 10.
```

Conditions are a special kind of expressions used in statements like `if` and `while` to control the flow of the execution of the program. In the above statements, often brackets are used to indicate it is a logical expression.

To test whether two variables are the same, you have to use two equal signs. Two equal signs will check the condition and one equal sign assigns an expression to a variable name. For "is not equal to" both != as well as <> can be used, but != is preferred. With combined conditions with many "and", "or" and "not" statements use brackets to avoid confusion:

```
not((x>xmin and x<xmax) or (y>ymin and y<ymax))
```

You can use logicals as conditions in if and while statements:

```
if inside:
    print("(x,y) is inside rectangle")


while not found:
    i = i + 1
    found = a==lst[i]
```

Note that `if inside` basically means: `if inside==True` and similarly `while not found` means `while found==False` .

2.9 List type: collection of items ordered in a table

2.9.1 What are lists?

Lists are not really an independent type but a way to group variables together. This allows you to repeat something for all elements of a list by using an index or by iterating through the list with the for-statement. This could be an operation, a search or a sorting action. Often it is useful to have a series of variables. Look at it as a table. An element of a list could be of any type, it can be an integer, float, logical, string or even a list! Special for Python is that you can even use different types in one list; in most programming languages this would not be possible. Both to define a list, as well as to specify an index, square brackets are used as in strings.

```
a = [ 2.3 , 4.5 , -1.0, .005, 2200.]
b = [ 20, 45, 100, 1, -4, 5]
```

```
c = ["Adama","Roslin","Starbuck","Apollo"]
d =[["Frederik",152000.], ["Alexander",193000.],
     ["Victor",110000.]]
e = []
```

This would make the variable **a** a list of floats, **b** a list of integers, **c** a list of strings. A list of lists as defined in `d` is basically a way to create a two-dimensional list. Finally `e` is an empty list. Accessing elements from the list is done as indicated below. Another way to make a list of numbers is using the so-called range function. This is an iterator function used primarily in loops, but it can also be converted to a list. The range function can contain one two or even three arguments:

```
range(stop)
range(start,stop))
range(start,stop,step))
```

In all these cases *start* is the start value (included), *stop* is the value for which the block is not executed because `for` will stop when this value is reached. So it is an inclusive *star*t but excludes *stop*. Another option is to enter a third argument, which is then is the step. The default start is 0, the default step is 1. Note that **`range( )`** only works with integers as arguments:

`list(range(5))` gives `[0,1,2,3,4]`
`list(range(5,11))` gives `[5,6,7,8,9,10]`
`list(range(5,11,2))` gives `[5,7,9]`

2.9.2 Indexing and slicing, list functions and delete

Let's use the following list:

```
lst = [1,2,3,4,5,6,7,8,9,10]
```

If we now print elements we can use the indices in many ways. Using one index, indicating a single element:

`lst[0]` which holds value 1

`lst[1]` which holds value 2

`lst[9]` which holds value 10

`lst[-1]` last element, which holds the value 10

`lst[-2]` one but last element which holds the value 9

Here it can be seen that indices start at 0, just as with strings. And similar to strings, the function len( ) will give the length of a list and thus the not to reach maximum for the list. Slicing lists with indices also works just as for strings, with three possible arguments: start, stop and step. Only one index refers to one single element. Two arguments separated by a colon refer to start and stop. A third can be used as step. If you look at the lists that were made in the previous paragraph about lists you can see that:

`a[1]` will return a float `4.5`
`b[1:4]` will return a list `[45, 100, 1]`
`d[1][0]` will return `"Alexander"` (do you see the logic of the order of the indices for the different dimensions: second element of main list, then first element of that list?)

Adding and removing elements to a list can be done in two ways the "+" operator or the append/extend functions:

`a = a+[3300.]`	will add 3300. at the end of the list
`a = a[:-2]`	will remove the last two elements of the list (by first copying the complete list without the last two elements, so not very efficient for large lists)
`a = a[:i]+a[i+1:]`	will remove element i from the list, when it's not the last one in the list
`a = b + c`	will concatenate (glue together) two lists if b and c are lists

Another way is to use the `del` (delete command) and/or functions which are a part of the list class. You call these functions by *variablename.functionname*( ) so a period between *variablename* and *functionname*. Some examples:

a.append(x) add x as a list element at the end of the list named a, equivalent with:
a = a + [x]
a.extend(b) extend the list with another list b, equivalent with:
a = a + b
a.remove(3300.) removes the first element with value 3300.
del a[-1] removes the last element of a list named a
del a[3:5] removes the 4th till the 6th element of a list named a
a = a[:3]+a[4:] will remove the 4rd element of the list named a

Slicing
The last example line uses the slicing notation (which can also be used on strings!). Slicing, or cutting out parts of a list is done with the colon. The notation is similar to the range arguments: *start*:*stop* and optionally a *step* can be added as third: *start*:*stop*:*step*. If no value is enter as start, the default value is zero. If no value is added as end the default value is the last. De default step is 1. Negative values can be used to point to the end (-1 = last element, -2 is one but last etc.).

Using the original definition of lst this will give:

```
lst = [1,2,3,4,5,6,7,8,9,10]
```

lst[:3]	first three element index 0,1,2: [1,2,3]
lst[:-2]	all except last two elements
lst[4:]	all elements except the first four: except elements nr 0,1,2,3
lst[::2]	every second element so with index 0,2,4, until the end

Other useful functions for lists:
b.index(45) will return the index of the first element with value 45
len(d) will return the length of the list
min(a) will return the smallest element of the list variable a
max(a) will return the largest element of the list variable a
sum(a) will return the sum of the elements in the list variable a

2.9.3*** Lists are Mutable: risks with multiple dimensions list creation

You can quickly build a one dimensional list with 5 elements like this:

```
>>> a=5*[0]
>>> a
[0, 0, 0, 0, 0]
>>> a[1]=1
>>> a
[0, 1, 0, 0, 0]
>>>
```

Warning: This only works with one dimensional arrays, see what happens when you try it with lists of lists, so two dimensional lists:

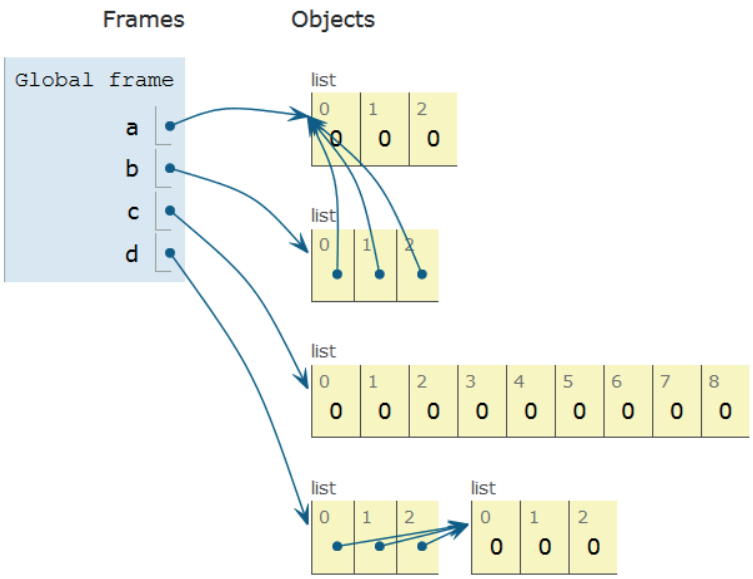
```
>>> a=3*[3*[0]]
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> a[0][1]=1
>>> a
[[0, 1, 0], [0, 1, 0], [0, 1, 0]]
>>>
```

So we only change the second element of the first row, but surprisingly all rows (all lists) have their second element changed into 1!

In general you could say, simply avoid this. But for the people who really want to know what is going on: In www.python.tutor.com we can see the difference. Imagine we will run the following code:

```
a = 3*[0]
b = 3*[a]
c = 3*a
d = 3*[3*[0]]
```

Then in the computer's memory, this is the result:



When using one dimensional lists with the asterisk (star) (so for example `*a` or `*[0]`), it will be an expression, evaluated and taken as a **value** and hence create a **new list**. Examples of this are how the list a and d are created. But normally in Python a list is seen as an **object,** which means it will just be copied by reference (a pointer to the same object, to the same memory location) will be used. So wo variables can point to the same object, hence the same memory location. This can cause behaviour which may seem strange at first sight.

To avoid this confusion (and ignore this difference), it is better to build a two dimensional array with append and values in a for-loop:

```
a=[]
for i in range(3):
      a.append(3*[0])
a[0][1] = 1
print(a)
```

Which will result in the following output:

```
[[0, 1, 0], [0, 0, 0], [0, 0, 0]]
```

But this code:

```
a=[]
b=[0,0,0]
for i in range(3):
      a.append(b)
a[0][1] = 1
print(a)
```

This only slightly different code will result in the problematic output again:

```
[[0, 1, 0], [0, 1, 0], [0, 1, 0]]
```

So 3*[0] is an expression, just like integer variables and float variables. But because b is a list-object and passed on by reference to the memory location, later referencing **a** will affect **b**.

In general, using append with non-list arguments is a safe method. This is because most variable types (floats, integers, strings) are passed on as value. They are called **immutable.** Expressions are also passed on as a value. List objects (like most objects) are called **mutable** and will be passed on as a reference to a memory location and hence can be changed by the function that is called.

Here are two ways to quickly create a two-dimensional list without the risk of creating pointers to the same list:

First create a one dimensional list, then replace each element by a list:

```
a=3*[0]
for i in range(3):
     a[i]=3*[0]

a[0][1] = 1
print(a)
```

Or with nested loops, create a fresh, new list (as a row) and then append this:

```
a=[]
```

```
for i in range(3):
    row = []
    for j in range(3):
        row.append(0)
    a.append(row)
a[0][1] = 1
print(a)
```

Both will give this output, indicating that a indeed consists of three independent lists:

```
[[0, 1, 0], [0, 0, 0], [0, 0, 0]]
```

So creating the variable `row`, ,or the list-element a[i] every time from scratch, is the safest way to avoid all appended lists pointing to the same memory location. If you are not sure, test a simple example in the above way: change one element and see the effect by printing the list. Or use www.pythontutor.com to see what is going on in the computer's memory.

2.9.4 List methods

An advantage of lists are the methods that come with it. These are functions which you call with a period after the variable name (just like append). An example is the sort function:

```
>>> a=[2,10,1,0,8,7]
>>> a.sort()
>>> a
[0, 1, 2, 7, 8, 10]
```

As we can see the function sort() (recognizable as a function because of the brackets) sorts the list a. This is a special function in that it changes the actual source list instead of creating a sorted copy. Most methods will give the result of the function and leave the original intact. Example are the method index and count. Index will give the index of the first occurrence of the given value. Count will give the total number of occurrences:

```
>>> a = [3,1,8,0,1,7]
>>> a.count(1)
2
>>> a.index(1)
1
>>> a.index(9)

Traceback (most recent call last):
  File "<pyshell#23>", line 1
    a.index(9)
ValueError: 9 is not in list
>>>
```

How would you make a program which will print the index of the first occurrence but also avoids the error message if it is not found by simply printing -1?

Exercise 1:
Make a program which calculates effect of a yearly the interest rate (entered by the suer) on start sum (also entered by the user). Print the amount after each year.

Exercise 2: Make a list of e.g. airplane companies by using the append( ) function, then print them below each other with the length of each word.

Exercise 3: You can also make a list which consist of different list, each with a different length, as shown here:

```
values = [[10, 20], [30, 40, 50, 60, 70]]
```

Print the length of each list and all the values it contains

2.10 Some useful standard, built-in functions

As we have already seen in the examples, Python has a number of built-in function which you can use in the right side of assignment statement on any expression using values and variables:

float(i)	converts an integer or string(text) to a float
int(x)	converts a float to an integer (whole number rounded to lowest integer)
round(x)	rounds off to nearest round number and converts to integer
round(x,n)	rounds off a float n decimals, result is still float
str(x)	converts a number into a string(text)
eval(txt)	converts a text to a number
bool(x)	convert x to a Boolean (typically non zero is True)
repr(x)	generates a string with a printable content of x
list(txt)	converts x to a list type (e.g. splits string in single characters)
chr(i)	gives character for ascii code i (look up ascii table to see codes)
ord(c)	gives acsii code of character c (a string with length one)
len(txt)	gives length of the string txt
len(lst)	gives length of the list lst
range(stop)	give a list [0,1,…stop-1] so stop is not included
range(start,stop)	give a list [start, startt+1,…stop-1] so start is included!
range(start,stop,step)	give a list [start, start+step,…] stop is not included
abs(x)	absolute value of x (modulo/length for complex numbers)
sum(a,b,c,d…)	total sum of variables a,b,c,d…
sum(lst)	total sum of the list elements in lst
max(a,b,c,d,.)	maximum value of a,b,c,d…
max(lst)	maximum value in the list lst
min(a,b,c,d…)	minimum value of a,b,c,d…
min(lst)	minimum value in the list lst

input(txt)	prints txt, asks user for input, returns user input as a text string
help(txt)	for use in shell: prints help on Python or installed function, module or statement, e.g. help(‘math’) or help(‘input’)

3. Python syntax: Statements

3.1 Assignment

In the previous chapter we'discuss the basic Python statements: commands that you use in a program which often do something with the data. A one page summary of the basic python statements can be found in Appendix A.

We already discussed the assignment statement at the beginning of the previous chapter. The assignment statement is one of the most simple statements in Python:

variablename = expression

In this statement the expression on the right hand side is evaluated and the stored in the variable with the name as given left of the equal sign. The *expression* determines the type and value of the variable and the left side, the *variablename,* the name.

It is possible to overwrite a variable, even when it is used in the expression on the right side. This is because the computer first evaluates the expression and only then stores the result in the variable, overwriting anything that was previously stored in that variable.

Some examples of how you can use the assignment are given below:

`a = 4.5`	will create a float with name `a` and value 4.5
`i = 100`	will create an integer with value 100 and name `i`
`total = 0`	will create an integer with value 0 and name `total`
`i = i + 1`	will increase the value of `i` with 1
`i += 1`	equal to previous line
`total = total + i`	will add i to the variable `total`
`total += i`	equal to previous line
`txt = "ea"`	will create a string with name `txt` containing "ea"
`txt = txt + "sy"`	will change it into "easy"
`serie = []`	will create an empty list
`serie = serie + [i]`	will add an element with value `i` to the list serie

Here it should be noted that a statement as i=i+1 may seem mathematically incorrect, but in programming this simply means that the expression i+1 is evaluated with the old value of I and the resulting value is then stored in i as this is the variable on the left side of the equal sign. So '=' means 'will become' instead of 'is equal to' (for which we use '==' in Python, for example when we test a condition in an if-statement). Also note the short-hand notation "+=", which has been discussed extensively in section 2.1 already.

3.2 Print statement

The `print` function generates text output in the console. It can print texts, variables and expressions. To print several things on one line use a comma to separate them. A space will be inserted automatically between the arguments. The space is the default so-called 'separator' symbol. After each call to print automatically a newline character is printed, so the next print call will start at a new line. A print-statement without any arguments, prints an empty line, or a newline character to be exact.

As example, three ways, which will result in the same output, it will print `Hello world` on one line:

```
print("Hello world")
print("Hello","world",sep=""))
print("Hello","world",end="\n")
print("Hello",end="")
print("world")
```

The above examples also use the use of the 'sep' and 'end' keyword, which can be used to use a different separator character or a different end of line.

The print statement can print one or more texts, variables expressions and the outcome of functions. Basically anything can be printed. Apart from generating output for the user, print is also very useful when trying to find bugs in your program. Every experienced programmer puts in a temporary print statement after each ten lines of code to check if everything works as intended. These are then later deleted or commented out with the hash symbol.

Other examples of valid print statements with other arguments than strings:

```
print("Program to solve 2nd order equations. ")
print("The solutions are",x1,"and",x2)
print("Discriminant is",b*b-4.*a*c)
print("List of names",lst)
print()
print("Ready. ")
```

Note: Compared to Python 2, PRINT has been changed from a statement into a function, this means the syntax has become different (although the Python3 syntax with the brackets will also work in the latest versions of Python 2):

Python 3 syntax: `print("The solutions are",x1, "and",x2)`
Old Python 2 syntax: `print "The solutions are",x1, "and",x2`

Next to 'sep=' and 'end=', there is also the keyword 'file=', which can be used to print to a file instead of to the screen. This file needs to be opened first with the open function. This function returns a file-object. The next example shows how this can be used to print to:

```
f = open('testje.dat','w') # Open file for writing (see ch 8)
print("abc",file=f)
print("def",file=f)
f.close() # Close file
```

Note how the hash symbol (#), indicates the start of the comments, this symbol and everything after it is ignored by the Python interpreter whrn the program is executed.

3.3 Input function

With the function **`input( )`** the user can be asked to input values during runtime of the program. Input is not really a statement, but a function in an assignment line. It returns the text as entered by the user during runtime. As input argument, it takes one string, which is used as the prompt by the function: it will print this text just before the user can give the input. Some examples:

```
name  = input(" What is your name: ")
ans   = input(" Do you want to continue? (Y/N) ")
```

The input function will always return a string. When another type of variable is needed as input, the converter functions can be used to get the right type

```
x = float(input("Give value for x: "))
i = int(input("Give number of values: "))
```

In some cases this might not work, will not work when a list of values is entered by the user, then the alternative method of using the evaluation function **`eval()`** is an option.This call to **`eval()`** will evaluate any expression and determine the type in the same as the assignment function would do it:

```
xlst  = list(eval(input("Give values for x:")))
a,b,c = eval(input("Enter a,b,c: "))
```

The eval function could also be used in the examples above, but there is a risk of the type ending up different from what you expected. Users will not always add a decimal point when entering a float for instance.

In Python 2 the input function would always call eval( ) , which means a separate function (called raw_input()) was required to avoid having to enter quotes when a string was entered. In Python 3 life has been made simpler by making input do what raw_input did in Python 2, and always returns a string. In this way the user can protect the input and convert it in a specific way.

Python 3.x:
`input` (*prompt*) => always returns a string, so needs conversion for numbers

Old Python 2.x syntax:

`input` (*prompt*) => use for **integer, float**, etc. type determined by user, equivalent to what **`eval(input(prompt))`** does in Python 3
`raw_input` (*prompt*) => returns **string,** just **`input( )`** does in Python 3

3.4 If-statement

The if statement has the following syntax:

```
if  condition :
        statement 1
        statement 2

statement 3
```

In this example above, only if the condition is *True*, the statements *statement 1* and *statement 2* are executed. If the condition is *False* it will automatically jump to *statement 3* without executing *statement 1* and *statement 2*. The conditional lines of code are marked byincreasing the margin, which is called **indenting** the lines after the colon. Lines with the same indentation level, so the same margin, is called a block of code. In Python a code block is marked only by this same margin. One indentation step is set to four spaces by default in IDLE but any number of spaces will do, as long as all statements in one block have the same indentation. In many other languages, you need extra brackets { } or being end statement to mark a block of code.

Because *statement 3* starts at the beginning of the line again, so un-indented, Python knows that this is the end of the conditional block. One of the statements inside the if-statement could be another if-statement, which would mean that another set of conditional statement would be indented one more time (nested if-loops).

Optionally the if-statement can be expanded with an `else`-branch or one or more else-ifs which are spelled as `elif` in Python. As many `elif`-branches can be added as necessary. Only one `else` can be added, always at the end of the `if` branche(s).

An example:

```
if x<xmin:
      x = xmin
      vx=-vx
      print(" Ball left via left side of screen. ")

elif x>xmax:
      x = xmax
      vx = -vx
      print("Ball left via right side of screen.")
else:
      print("Ball still on screen.")

x = x + vx*dt
```

In this example the condition `x>xmax` is only checked if `x<xmin` is False. Which probably is not a problem in this case. In the above example the assignment statement to update the variable x is always executed because it is not indented.

3.5 For-loop

The for loop is a very convenient loop, especially when it is used in combination with the `range()` function. For example:

```
for i in range(10):
     print(i)

print("Ready. ")
```

If you try this code, you will see that it prints the numbers 0 to 9. The range(10) function results in a list: [0,1,2,3,4,5,6,7,8,9] through which the variable *i* is iterated. This means: The indented block of code (one statement in this case) is executed ten times, once for each value of i. Another example:

```
total = 0

for i in range(1,11):
     print(i)
     total = total + i*i

print(total)
```

In this example, a variable `total` is initialized as zero just before the loop. But then in every execution of the loop `i*i` is added to `total`. Since `i` runs from 1 until but, not including, 11 in this code, the result is that `total` will be equal to 1*1 + 2*2 +3*3+ …+ 9*9 + 10*10 .

Instead of using `range( )` to generate a range of integers, any list can be used:

```
fellowship = ["Frodo","Sam","Merry","Pippin","Gandalf", \
              "Legolas","Gimli","Aragorn","Boromir"]

for name in fellowship:
     print(name)
```

If you need to loop floats e.g. from 0. to 10. with step of .1, you can also use the following construction. This example will print these values:

```
for i in range(101):
     x= i*0.1
     print(x)
```

When setting up loops (while or for), use print statements first (e.g. with a small number of iterations) to see whether your loop does what you intend it to do.

Tools for for-loops: Itertools
The module **itertools** has a lot of iteration tools which are very useful in cobinaiton with a for-loop. It can be used to get any permutation or combination of elelements for a loop. This can save a lot of programming time compared to when you would have to make that yourself.

For example, this bit of codeuses the permutations iterator function, which takes a list as input:

```
from itertools import permutations

lst = [1,2,3]

for a in permutations(lst):
    print (a,end=" ")
```

Will generate this output:

```
(1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)
```

And to take all possible selection of n combinations out of a list *lst*, the function combinations(*lst* , *n*) can be used:

```
from itertools import combinations

lst = [1,2,3,4,5]

for a in combinations(lst,2):
    print (a,end=" ")
```

Which will print:

```
(1, 2) (1, 3) (1, 4) (1, 5) (2, 3) (2, 4) (2, 5) (3, 4)
(3, 5) (4, 5)
```

There are many more useful funcitons in itertools, like cycle, product, etc. In the python shell, type help("itertools") to see the complete overview.

Exercise 1: Make a program that prints all the prime numbers till a certain value, (recall that a prime number is not divisible by any number except itself and 1).

Exercise 2: Write a Python program to construct the pattern on the right.

Exercise 3: Write a Python program that accepts a string and counts the number of digits and letters in the string.

3.6 WHILE-loop

A more versatile loop is the while-loop. It is a sort of a combination between an if-statement and a for-loop. A block of statements is executed for as long as a given condition is *True*. The block is repeated until the condition becomes *False*. The syntax is:

```
while condition:
        statement 1
        statement 2
        statement 3
```

The while is just like the if, used to control the flow of a program based on a condition. The difference is that in this case it is automatically repeated.

An example:

```
h0 = [0.0, 11000., 20000., 32000., 47000., 51000., 71000.]
hmax = h0[-1]
h = float(input("Enter altitude"))
h = min(h,hmax)
i= 0
while h>h0[i+1]:
    i = i+1
```

When you use this as a sort of "repeat until", you have to prepare the condition which is tested by while in such a way that the first time it will be true, so the loop as entered at least once. You can see this in the second example below, where found is set to False first.

Another example:

```
import math

n = int(input("Give a value of n: "))
i = 1
vn = int(math.sqrt(n))

found = False

while i<= vn and not found:

      if n%i==0:
            found = True

      i = i+1

if found:
      print(n, "is divisible by",i)
else:
      print("No factors of",n, "found.")
```

While can also be used to wait for a valid answer when using input:

```
choice = -1
while not(0 <= choice <= 3):
    choice = int(input("Enter your choice (1,2,3 or 0):"))

print("The user chose",choice,".")
print("Computer says: no.")
```

In this example the computer will keep asking the question until a valid choice 0, 1, 2, 3 is entered.

The two most important things need to be addressed when using while-loops:

- Often (not always) an **initialization** is required to make sure that the first time the condition in the while-loop is *True*, such that the while-loop starts running.

- Within the while- loop, the block of code has to assure that the condition becomes *False* at some point, avoiding an **endless loop**.

3.7 Loop controls: Break and Continue

Even though it is not considered a good programming style to use it as regular way to control the flow of your program, it is possible to make exceptions to the normal execution of the for- or while-loop. Two statements can do this: **`break`** and **`continue`**.

The command line **`break`** will break out of the current loop and continue with the next statement after the loop. The command line **`continue`** will also skip the remainder of the block of code in the loop but in contrast to the break command, it will return to the beginning of the loop to the test in the while-line (in a **`for`**−loop it will thus return to the beginning of the loop, taking the next value of the iterator).

Example of the `break` command within a while-loop:

```
while True:
    …
    if keypressed == ESCkey:
        break
    …
```

Example of the **`continue`** command within a for-loop:

```
for i in range(100):
    …
    if i%4==0:
        continue
    …
```

Note that **`continue`** and **`break`** can be used in both loop-types: so both can be used in for- and while-loops.

Using these two commands on a regular basis is considered to be a bad programming habit, as a clever definition of the loop can avoid these unpleasant disruption of the flow of the program. Also, take into account that an *if-then* statement could also do the job. The *break* and *continue* statements however, can make the code more readable since less indentations are required compared to the *if-then* statements.

Summarizing the difference:

`continue` skip the rest of the code in the block inside the loop and go to the line with the while or for statement. So in case of a for-loop the next value of the list will be used.

`break` jump out of the loop and proceed with the code after the loop

Programming language purists do not like these commands as they disrupt the flow of a program and can be used to create real "spaghetti-code", so only use them when there is no other neater way to achieve the same goal, e.g. to avoid too many indentations in the margin.

Exercise 1: We can make our prime number generator faster by means of the break command, make again a prime generator till a certain value. But now 'break' when a value is devisable by another value then itself or 1.

4. Making your code reusable and readable

When writing a piece of code it is important to make sure your code looks understandable to the outsider or to yourself when you look at the code in one year from now. You do not want to write code that you can only use once, so get used to this and do it all the time. Some suggestions:

- At the top of your script, write a **small summary** explaining the function of your code using comment lines. Also add your name and the date of editing.

- Use **empty lines** to create "paragraphs" indicating which parts belong together. Also use **spaces** in your line, they have no effect during runtime but can increase the readability

- Also use a **comment line** just before such a piece of code indicating in plain language what the meaning of the code is. Comments can be inserted when you start the line with a hash-character: #. The hash-sign can also be used to add comments at the end of the line (such as units of variables). An example:

```
#Initialize atmospheric values at sea level
p0   = 101325.          # [Pa]
rho0 = 1.225            # [kg/m3]
T0   = 288.15           # [K]
```

If you have a line which becomes very long, continue on the next line. Often Python will understand this because you are in the middle of a list or function call, but otherwise you may end the line with a backslash as a **continuation character**. Then Python will know that the next line needs to be added behind this line before it executes the command.

```
p0 = [101325.,  22632., 5474.9, 868.02,\
        110.91, 66.939, 3.9564 ]
```

Tip: Since it is often not very motivating to add comments afterwards, you can also do it in the other way around: When you have to design an algorithm, you first have to decompose the problem into smaller steps. Use comment lines to break your problem down in smaller steps and write in natural language what each step should do. Then all you have to do is fill in the gaps between the comment lines with the actual code and you have developed and commented your source code at the same time.

An example of the same code, both badly and well formatted, is given on the next page (**`pass`** is a statement which does nothing, sometimes used temporary before filling in final code):

```
File  Edit  Format  Run  Options  Windows  Help
def abc(a,b,c):
    d=b*b-4*a*c
    if d<0. or a==0.:
        return []
    elif d>0.:
        return [(-b-sqrt(d))/(2.*a),(-b+sqrt(d))/(2.*a)]
    else:
        return [-b/(2.*a)]
```

:

```
File  Edit  Format  Run  Options  Windows  Help
def solveabc(a,b,c):

# Function solveabc solves quadratic equation:
#                a x2 +  b x  + c = 0 for x
#
# Input: a,b,c = coefficients of polynomials (floats or integer)
# Output: list with 0,1 or 2 solutions for x (floats)
#
# User should check number of solutions by
# checking length of list returned as result

# Calculate discriminant
    D = b*b-4*a*c

# Create empty solution space S
    S = []

# Disriminant larger than zero: two solutions
    if D>0.:

        x1 = (-b-sqrt(D))/(2.*a)
        x2 = (-b+sqrt(D))/(2.*a)
        S.append(x1)
        S.append(x2)

# Discriminant negative, no solutions: do nothing
    elif D<0:

        pass

# Else, only one solution in rare case that D is exactly zero
    else:
        x = -b/(2.*a)
        S.append(x)

# Return resulting solution space
    return S
```

In this case the function is extremely simple, but imagine a more complex function: You would never be able to understand it or use it again a few months or years later, when coded in the format of the first example. When you make a habit of doing this while you code, it not an extra effort and it even saves you the effort of memorizing which part does what and what the variables mean during the debugging.

The small summary at the beginning of your code is called the header of a source file. A very good habit is to add this header at the beginning of each Python file. In this you can quickly describe the purpose of the program or the inputs and output of a function for potential users. For this you often want to use multiple lines with comments. You could of course do that using the hash sign:

```
#
# Main module of BlueSky Open ATM Simulator
#
# Start this module to start the program
#
# Created by    : Jacco M. Hoekstra (TU Delft)
# Date          : September 2013
#
# Modification  :
# By            :
# Date          :
#
```

But another way to add multiple line comments is the triple quote:

```
"""
Main module of BlueSky Open ATM Simulator

Start this module to start the program

Created by    : Jacco M. Hoekstra (TU Delft)
Date          : September 2013

Modification  :
By            :
Date          :

"""
```

Using this format will also make sure that python recognizes this as documentation for the function and will return it if the help function is called for your function. This is called the docstring. See below for an example:

```
    def complex(real=0.0, imag=0.0):
        """Form a complex number.

        Keyword arguments:
        real -- the real part (default 0.0)
        imag -- the imaginary part (default 0.0)

        """
        if imag == 0.0 and real == 0.0: return complex_zero
```

(Also note how default values are given to the arguments when this function is defined. When no argument is give, these values will be used).

5. Using modules like the math module

5.1 How to use math functions

As we already saw in the first chapter, Python's mathematical functions are defined in a math module which is one of the many modules that comes with Python. The math module contains:

- math functions e.g.: sqrt() (=square root) sin() (=sine of an angle in radians), asin() (=arcsine or inverse sine, returns the angle again in radians),
- angle unit conversion functions: degrees( ) and radians()
- constants like pi and e.

If you use a function that has been defined in a module you first have to input that module once at the beginning of the python script file in which you use the functions. This is done by typing:

```
import math
```

When we want to use a function from this module, we type the name of the module, math in this case, in front of the name of the function followed by a period (.) followed by the name of the function. For example:

```
x1 = (-b + math.sqrt(b*b-4.0*a*c)) / (2.0*a)
BC = k * math.sin(a)
alpha = math.asin(BC/AC)
```

The math module also contains the constants pi and e. These are used in the same way as the functions, with the module name in front of the constant's name:

```
bdeg = b/math.pi*180.0
y = math.e**x
```

Note that with logarithmic functions log(x) means ln(x), so base e and not base 10! To use a logarithmic function with base 10, use the function log(x,10.) or the special function log10(x) .

As it may be cumbersome to type 'math.' before these very common functions there are ways to import the names from the module into the namespace of the module. This means you don't have to put 'math.' in front of the function names every time, but you can use the function names as if they were defined inside your module. There are two ways to do this. Look at the examples below.

Method 1 to avoid this is directly importing selected functions from the module math:

```
from math import asin,pi

alpha = asin(BC/AC)
bdeg = b/pi*180.0
```

or easier, just import all functions from the math module with the asterisk symbol:

```
from math import *

alpha = asin(BC/AC)
bdeg = b/pi*180.0
```

In the first example the names are explicitly imported. This has two advantages: the reader of the source can tell from which module which function was imported. This is important if you use more than one module. Secondly, you prevent problems, which might arise if you use a variable name which happens to be the same as some function somewhere in the imported module.

Still, if there is any module where you could use the asterisk-import method, it would be the math module, since the names in there are well-known. In many other languages, these math functions are also reserved names and part of the so-called 'namespace' of the language.

A nice feature of the "from … import" way of importing functions is the option to rename functions:

```
from math import sqrt as wortel

x = wortel(9)
```

(This example will result in x being a float with value 3.0 as the sqrt() function converts the type to a float.)

On the next pages an overview is given of the main functions in math. There are a few extra in the actual module; e.g. those functions show how numbers are represented on a binary scale. If you're interested in these, use help ('math') in the shell or look up the math module in the Python Docs via the IDLE Help menu.

5.2 List of math module functions and constants

Constants

`math.pi`	The mathematical constant *pi* (= 3.1415926535897931)
`math.e`	The mathematical constant *e* (= 2.7182818284590451)

Power and logarithmic functions

`math.exp(`*x*`)`	Return `e**x`.
`math.log(`*x*`)`	Natural logarithm of *x* (to base *e*), so ln *x*
`math.log(`*x*`,`*a*`)`	Logarithm of *x* to base *a*, so $^{a}\log x$ (equals log(x)/log(a))
`math.log1p(`*x*`)`	Natural logarithm of *1+x* (so to base *e*), but then accurate for *x* near zero.
`math.log10(`*x*`)`	Base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.
`math.pow(`*x*`, `*y*`)`	`x` raised to the power `y` (same as `x**y`)
`math.sqrt(`*x*`)`	Square root of *x*. So this calculates $\sqrt{x}$

Trigonometric functions

`math.sin(`*x*`)`	Return the sine of *x* radians.
`math.cos(`*x*`)`	Return the cosine of *x* radians.
`math.tan(`*x*`)`	Return the tangent of *x* radians.
`math.acos(`*x*`)`	Arc cosine of *x* (i.e. the inverse cosine), resulting angle is in radians
`math.asin(`*x*`)`	Arc sine of *x* (i.e. the inverse sine), resulting angle is in radians
`math.atan(`*x*`)`	Arc tangent of *x* (i.e. the inverse tangent), resulting angle is in radians.
`math.atan2(`*y*`, `*x*`)`	Same as `atan(y / x)`, in radians but now the result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x,y)` makes this angle with the positive X axis. The point of `atan2()` is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle.
`math.hypot(`*x*`, `*y*`)`	The Euclidean norm, `sqrt(x*x + y*y)`. This is the length of the vector from the origin to point `(x, y)`.
`math.degrees(`*x*`)`	Converts angle *x* from radians to degrees.
`math.radians(`*x*`)`	Converts angle *x* from degrees to radians.

Hyperbolic functions

`math.sinh(`*x*`)`	Hyperbolic sine of *x*.
`math.cosh(`*x*`)`	Hyperbolic cosine of *x*.
`math.tanh(`*x*`)`	Hyperbolic tangent of *x*.
`math.asinh(`*x*`)`	Inverse hyperbolic sine of *x*.
`math.acosh(`*x*`)`	Inverse hyperbolic cosine of *x*.
`math.atanh(`*x*`)`	Inverse hyperbolic tangent of *x*.

Number functions

`math.ceil(`*x*`)`	Ceiling of *x* as a float, the smallest integer value greater than or equal to *x*.
`math.copysign(`*x*, *y*`)`	Returns *x* with the sign of *y* (so `abs(x)*y/abs(y)`)
`math.factorial(`*n*`)`	*n!* or the factorial of n (in Dutch: faculteit n)
`math.floor(`*x*`)`	Floor of *x* as a float, the largest integer value less than or equal to *x*.
`math.fmod(`*x*, *y*`)`	Modulo `fmod(x,y)`, returns `x - n*y` for some integer *n* such that the result has the same sign as x and magnitude less than `abs(y)`. This function fmod() is generally preferred when working with floats, while Python's `x%y` is preferred when working with integers.
`math.fsum(`*iterable*`)`	More accurate summing for floats than sum( )
`math.modf(`*x*`)`	The fractional and integer parts of *x*. Both float results carry the sign of *x*.
`math.trunc(`*x*`)`	Return the `Real` value *x* truncated to an `Integer` (usually a long integer)

5.3 The module random

Another often used module is a module which generates (seemingly) random numbers. This can be used for random initial conditions, dice or noise etc.

To find more information on the functions in this module, go to "Python docs" in the Help pull down menu in IDLE, we type random in the Search window.

Clicking on the first result brings us to chapter 10.6 Pseudo random generator. This section discusses the functions inside the random module.

After a lot of complex information (which we skip) we also see some useful functions:

random.random() Return the next random floating point number in the range [0.0, 1.0)

random.randint(a, b) Return a random integer N such that a <= N <= b.

random.randrange([start], stop[, step]) - Return a randomly selected element from range(start, stop, step). This is equivalent to choice (range(start, stop, step)), but doesn't actually build a range object but will result in one 'random' integer.

5.4 Explore other modules

To see which other modules are installed, you can type help('modules') in the Shell, but with Python(x,y) installed this quickly becomes too much (and takes too long). More info on specific modules supplied with Python can be found in the Python documentation. For instance check out the simple modules like os, sys and time, or more advanced like urllib or Tkinter! You'll find a treasure box of goodies. This is what is meant with when they say Python comes "with batteries installed".

6. Defining your own functions and modules

Using and making functions can make programming and debugging a lot easier. This is because it will make your main code shorter. It is also very useful when using the same calculation multiple times or to use the preprogrammed calculations.

6.1 Def statement: define a function

With Python you can define your own functions and in this way basically extend the language with your own commands. The statement def is used to define a function. The syntax of this statement is as follows:

```
def functionname(arg1,arg2,arg3):

    statement1
    statement2
    statement3

    return resultvalue
```

Examples:

```
def fibo(n):

# Generate fibonacci serie until value n
    serie = [a,b]

    while serie[-1]<=n:
        x = serie[-2]+serie[-1]
        serie.append(x)

# remove last element which was larger than n

    del s[-1]

    return serie
```

Functions can be complete programs or simple one-liners. Another example with a very short function:

```
from math import sin

def f(x):
    return x*sin(x)
```

Be aware that the `def` statement does not yet execute the function, it merely defines it. The code inside the function will only be executed when you call the function, after it has been defined

with def. So always call it, could be done manually from the interpreter to check the syntax and working of your newly defined function.

6.2 Multiple outputs

As you have seen in the formatting example, you can output several different values from your function. You can do this as a list, defined with the square brackets or as a tuple (a constant list) with the round brackets or a tuple by leaving out the brackets.

When you call the function you can store the result in a list/tuple or in different variables. The latter is done in the example below, which explains how to output more than one value of a function.

```
# multiple-returns.py
a, b, c = 0, 0, 0
def getabc():
    a = "Hello"
    b = "World"
    c = "!"
    return a,b,c #defines a tuple on the fly

def gettuple():
    a,b,c = 1,2,3 # Notice the similarities between this and getabc?
    return (a,b,c)

def getlist():
    a,b,c = (3,4),(4,5),(5,6)
    return [a,b,c]

# These all work, as amazing as it seems.
# So multiple assignment is actually quite easy.
a,b,c = getabc()
d,e,f = gettuple()
g,h,i = getlist()
```

6.3 Function with no outputs

When your function merely does something to the screen but there is no need to return a value, just use the `return` without a value. In the calling program, you can use this function just as you would use a statement.

```
# Function greet has no inputs or outputs

def greet():
    print("Hello, Alice.")
    return # This line could even be left out, indent is already marking end of block

# Print greeting from Exegesis by Astro Teller
# Calling the funciton only performs statements inside function
# These are not executed when the above code is read, as that part only
# defines the function

# The brackets are still needed so Python knows it is a function

greet()
```

6.4 Using one variable in definition as input and output is not possible

In some languages, a function can do something to the input variable without returning a value. This is not the case with the standard data types in Python, except for the list type.

In Python most data types are **immutable**, which means they are passed on as a value of the object, which is then used in the function. But since there is no reference to the memory location of the original variable in the calling module will not be changed by anything the function does. See the example below. Most types of variables (**floats, integers, strings, booleans**) are **immutable** because they are passed on by value. Others (lists) are passed on as reference to a memory location, then they would change as the function increases it. These are called **mutable** variables.

```
# Wrong way

def increaseit(a): # Here the input value is given the local name a
    a = a + 1      # This name is only valid inside the function
    return

a = 5
increaseit(a)
print(a) # This will print value 5

# Right way

def increaseit(a):
    return a+1

a = 5
a = increaseit(a)
print(a) # This will print 6
```

Mind you, this use of one variable as input and output at the same time in the definition of a function is considered very bad programming practice anyway in the procedural programming style in the languages where this is possible. Yet at the same time, it is common practice in object oriented programming, also in Python! It would then be called a method of the variable a and called as: **`a.increaseit()`**

6.5 Using functions defined in other modules: managing a project

As you have already seen with the math functions in the previous chapter, they are defined in a separate file(module) math.py and to use them, you first need to import this module.

You can also use this principle of defining functions in different files for large programs where you want to keep an oversight by storing different sets of functions in different files, or as we say in Python: in different modules. In this case the module file needs to be in the same folder so Python can find it.

mymain.py

```
import mytools

y = mytools.myfunc(x)
```

mytools.py

```
def myfunc(a):

 b = a*a-3.*a
 if b<0:
     b = a*a+3*a

 return b
```

Suppose you have defined a function `fibo` in the file `series.py` , and you want to use it in your own program `test.py` . Then you need to import your module `series`. We usually do this at the beginning of the source, since you need to do this only once. Because once the function is defined, you can use it as many times as you want. There are now many ways in which you can import and call this function:

Method 1:

```
import series
s = series.fibo(10)
```

Method 2:

```
from series import fibo
s = fibo(10)
```

Method 3:

```
from series import fibo as fibonacci
s = fibonacci(10)
```

Method 4 which I would discourage: you do not know which other names now suddenly have a special meaning, it also makes it harder to trace where functions are defined (I only use this with math and sometimes with Numpy for the math functions):

```
from series import *
s = fibo(10)
```

On the internet you can find many 3rd party modules which extend the functionality of Python to do a range of things. Once they are installed it is very easy to use them: just add an import call at the beginning of your script and all functions inside that module are available to you.

Exercises chapter 6

Exercise 1: Make a function that counts the words in a sentence and print the last word.

Exercise 2: Make four functions which calculate respectively the surface and volume of a sphere and the surface and volume of a cone.

Exercise 3:
Write a function:

def changestring(string, n, delim):

This function repeats string 'string' n times, separated by the string 'delim'.

7. Using logicals, example algorithm: Bubblesort

In this chapter to apply the logic and the loops we've described so far, we will have a look at the Bubblesort example. The first task is to generate a program that generates 50 random numbers under 100.

Here `randint`, see the chapter on the random module, can be used to make an unsorted list:

```
from random import randint

# Make an unsorted list
numbers = []

for i in range(50):
    numbers.append(randint(1,100))

print("Before sort:",numbers)
```

This prints an unsorted array of 50 random numbers under 100. Now we will sort it with the famous Bubblesort algorithm:

1. check all consecutive pairs in the list
2. if a pair is in the wrong order swap the places of the elements
3. keep checking all pairs until no swap is needed anymore

If we want to add this then we have to add the code on the following page:

```
# Make sure we enter while loop first time

swapped = True

while swapped:

# Now assume everything is done unless proven otherwise
    swapped = False

# Check all pairs by running index to one but last

    for i in range(len(numbers)-1):

# If they are in the wrong order: swap

        if numbers[i]>numbers[i+1]:
            a =numbers[i]
            numbers[i] = numbers[i+1]
            numbers[i+1] = a

# and set switch indicating that we swapped

            swapped = True


# Loop end when swapped is False and no swaps were performed anymore
print()
print("After sort:")
print(numbers)
```

Let's investigate this code:

The key line is this:

```
if numbers[i]>numbers[i+1]:
    a =numbers[i]
    numbers[i] = numbers[i+1]
    numbers[i+1] = a
```

First we check whether these elements are in the wrong order. When they are the same we should not swap them (it would keep doing this forever then). If wrong, we save the value of numbers[i] in the variable a, then we overwrite this with the value of numbers[i+1]. And then we use our saved value in a to put in numbers[i+1] resulting in a swap of places.

For this to work we need to let i run from 0 till the last place minus 1. Otherwise the numbers[i+1] will result in an error. So the for-loop reads:

```
for i in range(len(numbers)-1):
```

Now all we need to do is add some logic to detect whether we are done. We can use a logical swap to detect any swapping. For this we first set it to False outside the for-loop and as soon as we swap two variables we set it to True.

Then we add a while loop around this which will keep repeating our for-loop until swap remains False and hence no swap was performed. To make sure we enter the while loop the first time, we have to set it to True outside the while loop.

This is the order in which this program was made: from the inside out. First putting in elementary operations and then using a logical to add some logic.

The end result is an already complex looking construction. Look at the three indent levels: while, for and if are inside each other or nested as we call this in programming.

8. File input/output and String handling

8.1 Opening and closing of files

To read from a file you first have to make it accessible for the program. For this you use the open-statement. It opens the file and connects a variable of the type 'file' to it. This variable is not the content of the file, it is merely a 'handle' to do something with the file:

```
f = open("test.dat","r")    # Open file for reading
g = open("test.dat","w")    # Open file for writing
h = open("test.dat","a")    # Open file for appending
```

As soon as your done with the file, you need to release it again with the close statement.:

```
f.close()
g.close()
h.close()
```

This will not only release the file for use by other programs but also empty any read/write buffers which may be in use and whose content might get lost if the program ends abruptly without closing (and releasing) the files.

8.2 Reading from a text file

You can use two functions to read from a file: `readline()` or `readlines()`. As the name suggests, the first reads only one line and returns one string containing the text of this line. The second function readlines(), reads all lines of the file and returns a list of strings, one string per line read.

You can use the file handle variable in a while-statement when reading files:

The easiest way is to store it in a list of lines directly and use this list in the for-loop:

```
# Read a file
f = open("test.dat","r")    # Open file for reading
lines = f.readlines()
f.close()

for s in lines:
      …. Do something with the string variable s……
```

The file object will return False when the end of the file is reached. This is an example of how you can read lines from a text file using this feature:

```
# Read a file
f = open("test.dat","r")    # Open file for reading

while f:
      line = f.readline()
f.close()
```

Or use **readlines()** with a for-loop:

```
# Read a file
f = open("test.dat","r")    # Open file for reading

for line in f.readlines():
      …. Do something with the string variable line……

f.close()
```

8.3 Writing to files

There are two ways you can write to a file: in the overwrite mode or in the append mode. When you open the file with the string 'w' as second argument, you open it in the overwrite-mode. So the existing content will be deleted and replaced by what you write to the file between the open and the close statement in the current program. When you use the file in to write in the append-mode, so opened with as second argument 'a', you start to write at the end of the last line of the file. This you can use for log-files or error messages or to collect all output data. It appends anything you write at the end of the file (more precisely: at the end of the last line in the file).

Some examples of how you can write to a file:

```
f = open("test.dat","w")    # Open file for writing

for i in range(len(xtab)):
     line = str(xtab[i])+","+str(ytab[i])+"\n"
     f.write(line)

f.close()
```

Note the newline character at the end of the line: `\n` .
(On Windows PCs, you will often find the so-called DOS newline characters: which is `+chr(13)+chr(10)` = CR LF = Carriage return + Line Feed).

Similarly as with readline() and readlines(), there is here also a writelines(), which will write a complete list of strings to a file. See the following example code:

```
a = ["The Expanse\n",
     "A TV-serie by SyFy set in a future in which\n",
     "mankind has started exploring the solar system\n",
     "beyond the planet earth.\n"]

f = open("expanse.dat","w")
f.writelines(a)
f.close()
```

8.4 Reading a formatted data file

Often text files are formatted in a way which inhibits a straight read into a float or integer variable. Extra comment lines, text on the same line are very useful when a human reads the file, but often a burden when you have to read it with a computer program.

Let's look at an example: a data file (text format), which is named "test.dat":

```
C--- Flight angles

alpha = 5.224 [deg]
beta =-0.0112 [deg]
 gamma= -2.776 [deg]
theta =       [deg]
```

Suppose we want the program to read this file and store the result in a two-dimensional list with strings for the label and floats for the value per variable. So in this example, the text file should result in the following list:

```
[['alpha', 5.2240000000000002],
 ['beta', -0.0112],
 ['gamma', -2.7759999999999998]]
```

Before starting to write the program, already note that:

- The program should recognize **comment lines** because these start with a C-character.

- When **no value** is provided (as in the example with theta), it should also not appear in the list.

- **Empty lines** should have no effect, similarly **whitespaces** in the lines should not be critical (see gamma in example)

As a first step, we read the lines into a list of strings (see previous chapter):

```
# Read a file
f = open("test.dat","r")    # Open file for reading
```

```
lines = f.readlines()
f.close()
```

Now we have to process these strings. And then find the labels, strip the superfluous characters and lines and read the values in the list. This is achieved by the following program, which we'll briefly discuss below.

```
# Check every line in the list lines
for line in lines:

# Make case-tolerant: make lower case
    txt = line.lower()

# Check for comment line or a blank line
    if txt[0]<>'c' and not txt.strip()=="":
        words = txt.split("=")

        label = words[0].strip() # remove spaces at the start
        value = words[1].strip() #  and end of string

# Lots of checks for "[ ]" and "\n" at end of line
        i = value.find('[')
        if i>=0:
            value = value[:i]
        else:
            if value[-2:]=="\n":
                value = value[:-2]

# if a value is given: read it
        if value<>"":
            x = float(value)
            print(label," = ",x)
            nlist.append([label,x])
```

The for-loop makes sure the indented block of code will be executed for every string in the list variable lines.

To prevent that any further checks will not work because a lower case 'c' is used or somebody starts every name with a capital character, we change the whole line to lower case with the string method lower. Since it is not a general function but a method of the string type, we use the *variablename.methodname*(*arguments if any*) notation. Similar to the "append" method with lists.

Before we read the data two things are done: it is checked whether it is not an empty line or comment line and then the line is split up and cut using a combination of the split-method, the find-method and slicing.

There is a long list of string methods which can be handy. A selection is shown on the following page.

8.5 List of some useful string methods/functions

A complete list, as well as more detail per function, can be found in the Python Docs in the chapter on built-in types (6.6.1) or by typing `help('string')` in the shell window.

These are methods, which are functions that are part of the type string. This means they are called in a special way: by using the name of the string input variable instead of *str*, so followed by a period and then the name of the function. As the string is an immutable type, so passed only by its value, these functions return something as output: e.g. an altered copy of the string, an integer value or a logical.

```
line    = line.strip()    # To change variable line itself
lowline = line.lower()    # Copy result to other variable
n       = line.count("b")# Result is an integer
```

str.count(*ch, i , j*)	Number of occurrences of substring *ch* in *str*[*i*:*j*], (using *i,j* is optional.)
str.find(*ch*)	Index of first occurrence of *ch* in *str*, return -1 if not found
str.index(ch)	Index of first occurrence of *ch* in *str*, (error if not found) See also rindex( )
str.replace(*old,new*)	Return copy of *str* with all occurrences of substring *old* replaced by *new*
str.join(*lst*)	Return one string, with the strings from the list, separated by the character in *str*
str.startswith(*ch*)	Returns True if string *str* begins with the substring *ch*
str.endswith(*ch*)	Returns True if string *str* ends with the substring *ch*
str.split(*sep*)	Returns a list of strings: the words in *str* as separated by *sep* substring
str.splitlines( )	Returns a list of strings: *str* is split using the newline character
str.strip(*ch*)	Remove character ch from begin and end of *str*, default *ch* is a space. See also rstrip( ) and lstrip( ) for stripping only one side of *str*
str.expandtabs(*n*)	Replace tab-character by *n* spaces. If *n* is not provided 8 spaces are used.
str.lower( )	Returns copy of string *str* with all alphabetic characters in lower case
str.upper( )	Returns copy of string *str* with all alphabetic characters in upper case
str.title( )	Returns Copy Of String *str* But Then In Title Case, Like This
str.capitalize( )	Returns copy of string *str* with the first character capitalized
str.islower( )	Returns True if all characters in the string *str* are lower case
str.isupper( )	Returns True if all characters in the string *str* are upper case
str.istitle( )	Returns True If The String *str* Is In Title Case, Like This
str.isalpha( )	Returns True if all characters in the string *str* are alphabetic characters
str.isdigit( )	Returns True if all characters in the string *str* are digits
str.isalnum( )	Returns True if all characters in the string *str* are digits or alphabetic
str.isspace( )	Returns True if all characters in the string *str* are whitespace

8.6 Genfromtxt: a tool in Numpy to read data from text files in one line

To use genfromtxt(), the Numpy module has to be imported (by the way, it is also included in Scipy):

```
import numpy as np
```

Reading data from text file with genfromtxt, example:

```
table = np.genfromtxt("test.dat",delimiter=",",comments="#")
x  = table[:,0]   # x in first column
y1 = table[:,1]   # y1 in 2nd column
y2 = table[:,2]   # y2 in third column
```

Here we see that the different columns of the two-dimensional array `table` are copied in independent one-dimensionnal arrays `x`, `y1` and `y2`.

From reference manual, all parameters of genfromtxt:

`numpy.genfromtxt`(*fname*, *dtype=<type 'float'>*, *comments='#'*, *delimiter=None*, *skiprows=0*, *skip_header=0*, *skip_footer=0*, *converters=None*, *missing=''*, *missing_values=None*, *filling_values=None*, *usecols=None*, *names=None*, *excludelist=None*, *deletechars=None*, *replace_space='_'*, *autostrip=False*, *case_sensitive=True*, *defaultfmt='f%i'*, *unpack=None*, *usemask=False*, *loose=True*, *invalid_raise=True*)

Loads data from a text file, with missing values handled as specified. Above the default values for the keywords are shown.

Each line past the first *skiprows* line is split at the *delimiter* character, and characters following the *comments* character are discarded.

On the next page a selection of useful parameters to use with genfromtxt.

A Selection of parameters to use with genfromtxt:

fname : file or str
File or filename to read. If the filename extension is gz or bz2, the file is first decompressed.

dtype : dtype, optional
Data type of the resulting array. If None, the dtypes will be determined by the contents of each column, individually.

comments : str, optional
The character used to indicate the start of a comment. All the characters occurring on a line after a comment are discarded

delimiter : str, int, or sequence, optional
The string used to separate values. By default, any consecutive whitespaces act as delimiter. An integer or sequence of integers can also be provided as width(s) of each field.

skip_header : int, optional
The numbers of lines to skip at the beginning of the file.

skip_footer : int, optional
The numbers of lines to skip at the end of the file

missing_values : variable or None, optional
The set of strings corresponding to missing data.

filling_values : variable or None, optional
The set of values to be used as default when the data are missing.

usecols : sequence or None, optional
Which columns to read, with 0 being the first. For example, usecols = (1, 4, 5) will extract the 2nd, 5th and 6th columns.

9. Matplotlib: Plotting in Python

9.1 Example: plotting sine and cosine graph

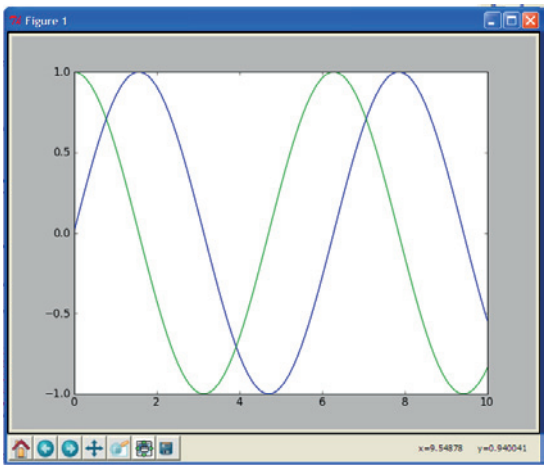
Making graphs in Python is very easy with the `matplotlib` module. A large range of types of plots can be made on screen and saved in several high-quality formats (like .eps) to use in reports etc.

Try the example program below (or download **plotsincos.py** from blackboard). The header is the standard way to import both matplotlib and matplotlib.pyplot. It uses `import` *module* `as` *newname* , to create shorter names, which make it easier to use the modules.

Pyplot is a module inside a module and imported as `plt` in the code below. Then it uses a for-loop to generate different values for x and with this, two y-values using sine and cosine are calculated. Append each value to the list variables xtab, y1tab and y2tab and then plot these in the same figure. This figure will only become visible when a call to plt.show() is made. This starts a separate program with the plot and waits until the user has closed this window.

```
from math import sin,cos

import matplotlib.pyplot as plt

# Goal: Draw a sine ans a cosine for x is [0,10] with 100 steps of 0.1

print("My first plot program")
print("Generating sine and cosine tables...")

# Generate lists with x and y values
xtab  = []
y1tab = []
y2tab = []

step = 0.1

# Note how we use an integer range function to generate floats

for i in range(101):
    x = float(i)*step+0.0
    y1 = sin(x)
    y2 = cos(x)

    xtab.append(x)
    y1tab.append(y1)
    y2tab.append(y2)

# Plot both lines in graph

plt.plot(xtab,y1tab)
plt.plot(xtab,y2tab)
plt.show()
```

If you run this program, you will see the following window appear as a result of plt.show():



Now try some of the buttons below the graph. From left to right these are:
- Home button: brings you back to default figure as shown in the beginning
- Back button: return to previous plot when you've panned or zoomed
- Next button: opposite of back
- Pan button: move the plotting window coordinates
- Zoom: Select a zoom window
- Adjust subplot: Adjust (sub)plot parameters with sliders
- Save: Select a file format and filename to save the plot as currently set up in the window

More options are available if we define a figure and a subplot. We can then use the methods provided in these types:

```
# Plot both lines in graph
fig = plt.figure()
sc = fig.add_subplot(111)

sc.plot(xtab,y1tab)
sc.plot(xtab,y2tab)
sc.set_title('Now a title can be added to a figure')

# When done plotting, show interactive plot window
plt.show()
```

The variable fig can now also be used to add different subplots next to each other in one figure. We define one subplot here which we call sc (sine-cosine) and then use methods provided with the plot to both plot the sine and cosine as well as add a title. Similarly functions for legends and labels are available.

A nice way to explore all options is to go to the **Matplotlib gallery**:

http://matplotlib.sourceforge.net/gallery.html

In this gallery, click on a figure and then select "source code" to see how the figure is made. Ignore all the complex stuff where they generate the data and check for useful Matplotlib

methods in a figure or a subplot to enhance the way your graphs look. Also the different types of graphs like bar charts are illustrated.

Exercise 1: *Make a plot of the two functions f(x) = sin x and g(x) = e^x – 2 for x = [0, π]. Find the coordinates of the point where sin x = e^x – 2 using the zoom-function of the plot window.*

Exercise 2: *With a simple cosine function you can already make beautiful flowers in python. So make a flower which has the function r = 4cos(2ϑ) in polar coordinates.*

Exercise 3: *Plot the following data to make it more readable:*

Time (decade)	*1950*	*1960*	*1970*	*1980*	*1990*	*2000*	*2010*
CO2 concentration (ppm):	*250*	*265*	*272*	*280*	*300*	*320*	*389*

9.2 More plots in one window and scaling: 'subplot' and 'axis'

Instead of several lines in one plot, you can also include more than one figure in one plot window with the subplot function. In the example below, we want to show the standard goniometric functions sine, cosine and tangent in the top row and the hyperbolic functions in the bottom row. This is done by including the command subplot. For example the first subplot-command:

```
plt.subplot(231)
```

The number 231 is used in an unusual way, it should be read as 2-3-1 and stands for: make 2 rows of 3 graphs and start with subplot number 1. Until the next subplot command, all function calls will now affect this subplot. The numbering of the plots is in this case 1 to 3 for the plots in the first row and 4 to 6 for the three plots in the bottom row.

Note that since the tangent had vertical asymptotes, it will yield a useless high range on the y-scale. To control the **range of the axes** from within our program for the third subplot, we use the command:

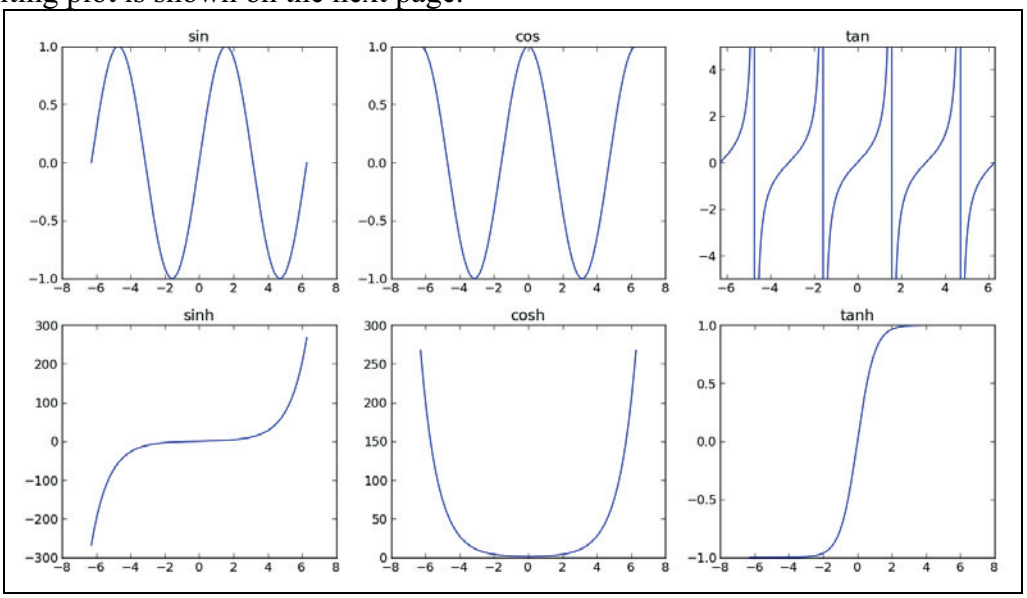
```
plt.axis([-2.*pi,2.*pi,-5.,5.])
```

This will set the x-axis at a range of [-2π, 2π] and the y-axis at [-5,5]. This shows the shape of the tangent in a more sensible way.

```
from math import pi
import numpy as np
import matplotlib.pyplot as plt

# Generate data

x  = np.linspace(-2.*pi,2.*pi,1000)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)

z1 = np.sinh(x)
z2 = np.cosh(x)
z3 = np.tanh(x)
```

```
# Make 2 rows of 3 graphs, start with plot 1

plt.subplot(231)
plt.plot(x,y1)
plt.title('sin')

plt.subplot(232)
plt.plot(x,y2)
plt.title('cos')

plt.subplot(233)
plt.plot(x,y3)
plt.axis([-2.*pi,2.*pi,-5.,5.]) # setting scale:xmin,xmax,ymin,ymax
plt.title('tan')

# Second row: plot 4-6

plt.subplot(234)
plt.plot(x,z1)
plt.title('sinh')

plt.subplot(235)
plt.plot(x,z2)
plt.title('cosh')

plt.subplot(236)
plt.plot(x,z3)
plt.title('tanh')

# Show plot window

plt.show()
```

The resulting plot is shown on the next page.



You can also see that in the plot the asymptotes are just nearly vertical jumps in the data. You could draw the tangent graph in such a way that it will not connect these points. You can even add drawing the asymptotes with red dashed lines when you know where they are. How would you do this? Check out the examples and the Matplotlib reference documentation for this. (Another way might be to draw an independent line in graph for each period!)

Exercise 1: *Plot two functions y1 = cos(3x) + sin(2x) and y2 = e^(2x) - e^(-3x+1) on the interval x = [0,10] both on a different graph and place them under each other.*

Exercise 2: *Plot $r = \vartheta$ and $r = \vartheta^2$ on the interval $\vartheta = [0,8\pi]$ in three graphs. Make sure that you label all your axis and give titles to the plots.*

Exercise 3: *Sometimes it is convenient to plot straight lines in your plot. It works almost the same as changing your axis system. You use the function plot( ):*
`plot([x1, x2], [y1, y2], color='k', linestyle='-', linewidth=2)`
Make a plot with a vertical, horizontal and diagonal line.

9.3 Interactive plots

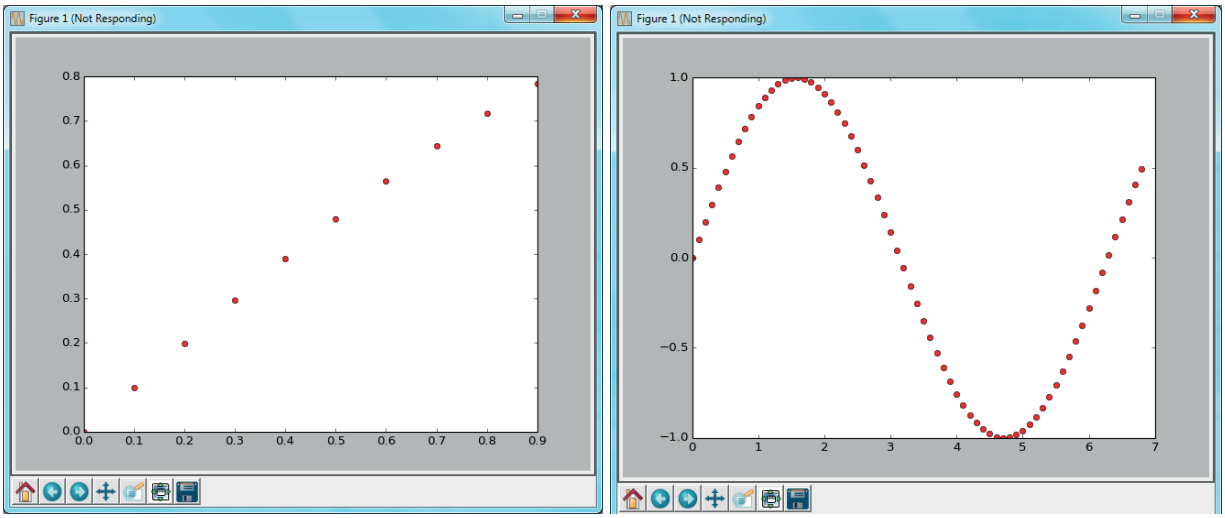
Sometimes you do not want to wait till your program is ready, but you would like to see the data already when it is being generated. For this you can use the interactive plot feature. By default this is switched off, but you can switch it on and off using the `matplotlib.pyplot.ion()` and `matplotlib.pyplot.ioff()` commands.

When done, close the plot with `matplotlib.pyplot.close()` to avoid stalling your computer (close is especially important on Windows PCs)

See for example the program below:

```
from math import *
import matplotlib.pyplot as plt

plt.ion() # Switch on interactive mode

for i in range(100):
    x = float(i)/10.
    plt.plot(x,sin(x),"ro")    # plot each point as a red dot
    plt.draw()                 # Show result

plt.close()
```

Which will show data as it is being generated and adjust axes on the fly when necessary:



Exercise 1: Plot the two functions $y = (4 - x^2)^{1/2}$ *and* $y2 = -(4 - x^2)^{1/2}$ *on the interval* $x = [-2,2]$ *as an interactive plot and see which shape appears.*

9.4 3D and contour plots

When you need to plot a function of two variables, for example z = f(x,y) , another type of graph may be needed. In that case there are two common ways to do this. Using colours indicating the z values per point x,y or using a 3D plot where the height indicates the value z. Both methods are explained in the following example codes.

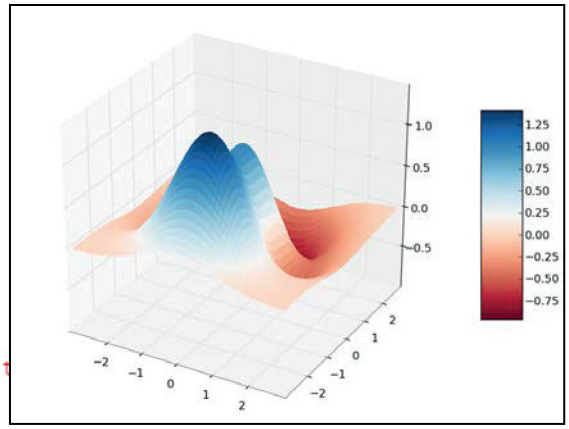
Both examples plot the function z=f(x,y) . For this they use a 2-dimensional array with a grid, generated with a function called “meshgrid”, to generate the values at an equal distance. You could convert these examples of source code to a function which plots your own data or your own function. You should then only change the data X,Y and Z. To do this in a neat way, you could convert these examples to a function where the X,Y and Z are input arguments to that function.

The resulting figure is shown next to the source. In the 3D plot you can move around the plot to change the viewing angle before saving it to an image file. Both plots also use a colour map cm.

plot3D.py - D:\Data\Python\Projects\AE1106\Examples\Block2\plot3D.py

File Edit Format Run Options Windows Help

```
from numpy import exp,arange,meshgrid,arange
import matplotlib.pyplot as plt

from matplotlib import cm
from mpl_toolkits.mplot3d import axes3d, Axes3D

# Define the function that we're going to plot
def z_func(x,y):
 return (1-(x**2+y**3))*exp(-(x**2+y**2)/2)

# Make the mesh and the z-value
x = arange(-3.0,3.0,0.1)
y = arange(-3.0,3.0,0.1)
X,Y = meshgrid(x, y) # grid of point
Z = z_func(X, Y) # evaluation of the function on t

# Create the 3D plot
fig = plt.figure()
ax = Axes3D(fig)

# 3D surface plot
surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                       cmap=cm.RdBu,linewidth=0, antialiased=False)
# Legend
fig.colorbar(surf, shrink=0.5, aspect=5)

# Show figure
plt.show()
```

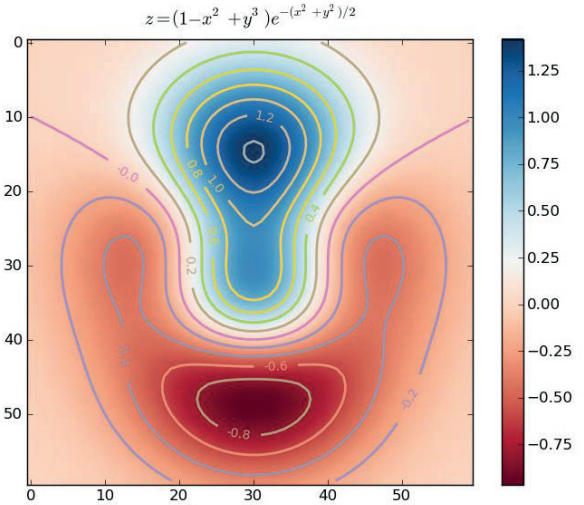


And for the colour- and contour plot we can use the following example code:

plotcontour.py - D:\Data\Python\Projects\AE1106\Examples\Block2\plotcontour.py

File Edit Format Run Options Windows Help

```
from numpy import exp,arange,meshgrid
import matplotlib.pyplot as plt

# Define the function that we're going to plot
def z_func(x,y):
 return (1-(x**2+y**3))*exp(-(x**2+y**2)/2)

x = arange(-3.0,3.0,0.1)
y = arange(-3.0,3.0,0.1)
X,Y = meshgrid(x, y) # grid of point
Z = z_func(X, Y) # evaluation of the function on the grid

im = plt.imshow(Z,cmap=plt.cm.RdBu) # drawing the function

# adding the Contour lines with labels
cset = plt.contour(Z,arange(-1,1.5,0.2),linewidths=2,cmap=plt.cm.Set2)
plt.clabel(cset,inline=True,fmt='%1.1f',fontsize=10)
plt.colorbar(im) # adding the colobar on the right

# latex fashion title
plt.title('$z=(1-x^2+y^3) e^{-(x^2+y^2)/2}$')
plt.show()
```



(Examples derived from examples at http://glowingpython.blogspot.com/ made by @Justglowing)

Exercise 1: The deflection of a wing can roughly be approximated by $z = (1/1000)x^2$, make a 3D plot of the deflection of the wing. The wing is rectangular with a span of 40 m and a chord of 8 m. Take the span as your x-direction, the deflection in the z-direction and the chord in the y-direction.

9.5 Plotting on a map

Matplotlib has a downloadable toolkit called basemap, which allows you to include a map as background of your plot. But much easier is using a Google maps wrapper called **pygmaps**. Pygmaps generates an HTML file with your data which you can show with your browser. Get the pygmaps.py file from: (https://code.google.com/archive/p/pygmaps/downloads). An examples and their output is shown below. It uses a file with flight data and plots this on a map.

```
"""
Show four flight log file on Google maps window
May 10th, 2014
Jacco M. Hoekstra
"""

from numpy import *
import pygmaps
import os

# Set window position and zoom level
# Define map object: lat,lon,zoom level
mymap = pygmaps.pygmaps(51.85 , 5. , 10)

# Read data, 2nd column ([1]) and third ([2]) are lat,lon in degrees
fdata = genfromtxt("flight1.log",comments="#",skiprows=2)

# Create path list: (lat1,long1), (lat2,long2)
path = list(fdata[:,1:3])

# Add path to map
mymap.addpath(path,"#FF0000")

# Make html file (comparable to pygame.display.flip() or plt.show() )
print "Showing map window..."
mymap.draw("test.html")

# Use os module to start html file
os.system("test.html")
print("Ready.")
```

#FF0000 means a RGB color code, it means 255,0,0 which indicates full signal on the red color, and none on the green and blue. So it is light, pure red. In pygame it would be (255,0,0), in some libraries colour codes are floats, then it would be (1.0,0.0,0.0)

9.6 Overview of essential matplotlib.pyplot functions

It is assumed you have imported matplotlib.pyplot as plt, so the pyplot commands are preceded by a "plt."

plt.plot(*x*,*y***)** Plot a line graph using points x[i],y[i]

`plt.plot(x,y,"r-")` Example of line formatting: red line, try also `"b+"`, `"go"`
`plt.plot(x,y1,label="y1")` gives name a label
`plt.legend()` shows a legend

plt.bar(*x*,*y***)** Bar plot, *x* = hor. place (e.g. arange(5), *y* , where y is alist/array with 5 values)

plt.show( ) Show graph in interactive window (put this at the end of your plot commands or you will not see anything)

plt.axis([*xmin*, *xmax*, *ymin*, *ymax***])** Set range of axes, uses a list (or tuple) with the values

plt.grid(True) Switch on grid (lines based on scale)

plt.xlabel(*str***)** Use text in *str* as label on x-axis
plt.ylabel(*str***)** Use text in *str* as label on y-axis
plt.title(*str***)** Use text in *str* as title on diagram

```
plt.xlabel('time (sec)',fontsize=14, color='red')
```
Example text formatting

plt.legend(loc=*locstr***)** Add legend, uses a list of strings: str1 is text for 1st line etc. Location of box with legend can be specified with this string:

"upper right"	"center left"
"upper left"	"center right"
"lower left"	"lower center"
"lower right"	"upper center"
"right"	"center"

plt.subplot(231**)** Specify number of rows (2), number of columns(3) and finally set the figure number for next pyplot commands (1st figure in this example)

plt.axvline(x=1) Draw a vertical line x=1. You can also limit the length with the ymin and ymax keywords: plt.axvline(x=1,ymin=0.,ymax=2.,color='r')

plt.axhline(y=2) Draw a horizontal line y=2. You can also limit the length with the xmin and xmax keywords: plt.axhline(y=2,xmin=-1.,ymax=1.,color='r')

10. Numerical integration

10.1 Falling ball example

The example assignment below describes a technique called *numerical integration*: for every time step, the change of a variable is first calculated and then used to change the value of that variable. It is called numerical *integration* because you move from the second derivative of place to time (the acceleration) to the first derivative of place to time (the speed) and then the derivative to the place itself (in this case the altitude). So those two steps *integrate* with respect to time.

In complex examples many forces can act on a mass, but you can often calculate these for a given situation. And with this technique, if the forces can be calculated for a situation, we can calculate the acceleration and then integrate this to get the time history of speed and place. In this way we can simulate the physics and solve problems, which we cannot solve analytically.

Try for instance to implement the simulation described below.

1. Simulate a falling mass using the following equations:

   a) Initialize a simulation with the following variables:

$t = 0.$	[s]	time starts at zero
$dt = 0.1$	[s]	time step
$vy = 0.$	[m/s]	starting speed is zero
$y = 10.$	[m]	start altitude
$g = 9.81$	[m/s^2]	gravity acceleration
$m = 2.0$	[kg]	mass

   While y remains larger than zero (until the mass hits the ground), repeat the next iterations:

   $t = t + dt$
   $F = m \cdot g$
   $a = - F/m$
   $vy = vy + a \cdot dt$
   $y = y + vy \cdot dt$

   During each step, append the value t to a table for the time and append the value of y to a table with the value of y. After all iterations, when the mass has reached the ground, plot the value of the height y against the time t, using these tables.

   b) Add the drag to the example. Use $D = C_D \, ½\rho V^2 S$ with:
   $C_D = 0.47$, $\rho = 1.225$ kg/m^3, $S = \pi R^2$ and $R = 0.15$ m

   c) Compare the two different datasets in one plot to see the effect of the drag. Try different values for starting altitude and C_D.

10.2 Two-dimensions and the atan2(y,x) function

Our example of the falling ball has only one dimension, but this can easily be expanded. Imagine a horizontal speed is added:

$$x = x + vx*dt$$

But when drag is then added things get more complicated in two dimensions.

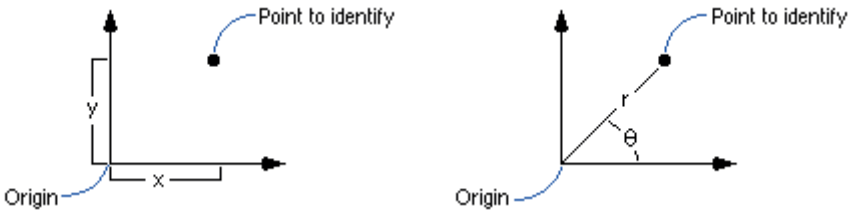
The drag can be calculated in the direction opposite of the speed, but then it needs to be composed in two directions. One way to do this is by calculating the length of the force and then decompose it into two directions.

Often it is handy to switch between polar coordinates for the positions and to orthogonal coordinates to get the forces. We can this use for calculating acceleration, speed and displacement in both x- and y-direction. For this there is a very convenient, special arctan-function called `atan2(y,x)`. Look at the example below:

```
V = sqrt(vx*vx+vy*vy)
D = cd*0.5*rho*V*V*S
angle = atan2(-vy,-vx)

Dx = D*cos(angle)
Dy = D*sin(angle)
```

Here we see that first the length of the speed vector is calculated. This is then used to calculate the length of the drag force. Then the angle is calculated using a special arctan-function atan2. If the normal atan function would have been used, there will be no different outcome for –vy/-vx and vy/vx even thought the two vectors point in opposite direction. This is caused by the fact that the tangent, being the quotient of sine and cosine, has a period of only π and not 2π.

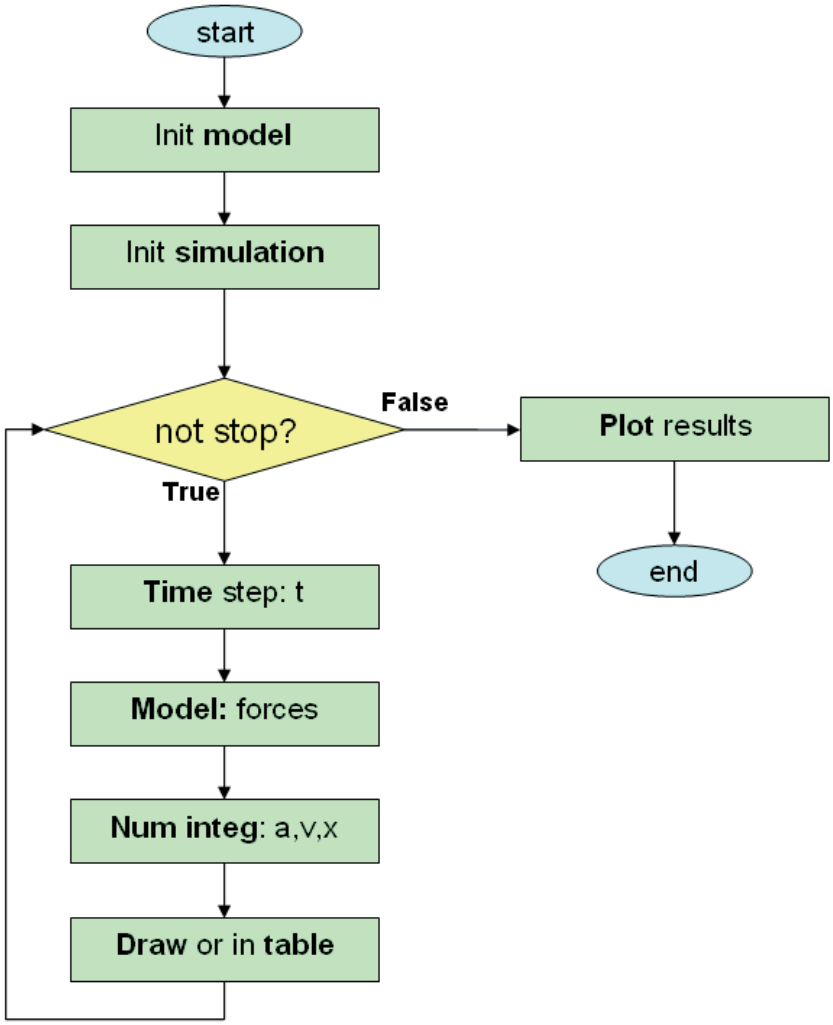
To be able to move from Cartesian coordinates (x,y) to polar coordinates (r,θ) without an extra if-statement for checking the sign of x and y, atan2 can be used as in the example. It will result the correct angle for the (y,x) (vy,vx) or (dy,dx). Since the tangent is sine/cos, hence y/x the y-coordinate is given first. So with this function θ=atan2(y,x) it works for all quadrants!



$$\theta = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{if } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ 0 & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$

10.3 Program structure of numerical integration and simulation

Note how each simulation has the same structure:



Use this structure as a template for all your simulations: add the descriptions in the block as comment lines and then fill these in with real code between the comment lines.

Chapter 10 exercises

Exercise 1:

a) Initialize a simulation with the following variables:

t = 0.	*[s]*	*time starts at zero*
dt= 0.1	*[s]*	*time step*
vy = 0.	*[m/s]*	*starting speed is zero*
y = 10.	*[m]*	*start altitude*
g = 9.81	*[m/s^2]*	*gravity acceleration*
m = 2.0	*[kg]*	*mass*

While y remains larger than zero (until the mass hits the ground), repeat the next iterations:

$t = t + dt$
$F = m \cdot g$
$a = - F/m$
$vy = vy + a \cdot dt$
$y = y + vy \cdot dt$

During each step, append the value t to a table for the time and append the value of y to a table with the value of y. After all iterations, when the mass has reached the ground, plot the value of the height y against the time t, using these tables.

b) Add the drag to the example. Use $D = C_D \, ½\rho V^2 S$ *with:*

$C_D = 0.47$, $\rho = 1.225$ kg/m^3, $S = \pi R^2$ *and* $R = 0.15$ m

c) Compare the two different datasets in one plot to see the effect of the drag. Try different values for starting altitude and C_D.

Exercise 2:

a) Make a small simulation of a cannon ball which has a mass of 10 kg, that is fired with a starting speed of 100 m/s, but it is shot at an angle of 30 ^{0}. Besides that, the canon is placed one meter above the ground and you can assume that this is also the starting height of the cannon ball. Show a plot of the vertical distance against the horizontal distance. So you can see the trajectory of the cannon ball.

b) Now extent this program, because this model has too many simplifications. For example a cannon ball is not frictionless, so we should include the drag. For now we assume it is constant and N is set to for example 80. Try to find the accelerations with force equilibrium and plot the trajectory of the ball.

c) The last step to finalize our program is that the drag is not constant. It is actually a function of speed, in this program use the relation that $Fd = 0.05V^2$. *From Intro I you know that the gravitational constant is changing with altitude, also include this effect in your program. Hint:* $g = g_0(R_{Earth}/(R_{Earth} + h))$, *where* $R_{Earth} = 6371$ km.

11. Numpy and Scipy : Scientific Computing with Arrays and Matrices

11.1 Numpy, Scipy

The modules Numpy and Scipy have provided users of Python with an enormous range of engineering and scientific computing tools. Many of this has been inspired by the functionality of MATLAB and its provided toolboxes. The syntax and names of functions are often identical.

Python with Numpy and Scipy is more capable than Matlab in many respects. Python is better in handling strings, reading files, working with very large projects and with large datasets. Python can also be used in an object oriented programming way. Both Spyder and the iPy Notebook provide a very user-friendly environment for scientists. A more general difference are the extra possibilities, which are provided by a full-featured general purpose programming language like Python. And, often more importantly, MATLAB is very expensive and many applications require extra toolboxes, which are in turn also very expensive. This also hinders sharing tools as well as quickly using source code from the internet community: often you can only use the downloaded bits after purchasing the required toolboxes.

With Numpy and Scipy, Python has surpassed MATLAB in terms of functionality. The modules are available for free (as are all Python modules). They are developed, maintained, expanded and used by a large academic community, mainly in the US and Europe.

Numpy forms the foundation for Scipy, Matplotlib and many other modules: it provides the array- and matrix-types as well as linear algebra functions as extension to Python. Scipy adds a whole range of sometimes very dedicated scientific computing and engineering functions. Thanks to Numpy and Scipy, Python has become the default language of choice for scientific computing, according to IEEE and many others. Let's explore the capabilities of these modules, even though we can only scratch the surface in this course.

To use these modules, they need to be imported. It has become standard practice to rename them when importing them. Numpy becomes "np" and Scipy becomes "sp". This means we will often see the following header in our scientific applications. In this course, we assume you have imported the modules as follows:

```
import numpy as np
import scipy as sp
import matplotlib as mplt
import matplotlib.pyplot as plt
```

In the Numpy and Scipy documentation it is often even assumed that you have imported everything using `from numpy import *` . So do not forget to type `np.` before the Numpy functions and "`sp.`" before the Scipy functions, even though you don't see this in the Numpy and Scipy documentation. Most Numpy functions can also be used as if they were a part of Scipy, but then with the "sp." prefix.

This would also be a good time to add a link to the Help chm-files containing the Numpy and Scipy reference to your IDLE Help menu, if you have not already done so. Go to Options>Configure IDLE, click on the "General"-sheet on the top right. Then click "Add" in the lower part of the window to add the menu item and the link to these files.

11.2 Arrays

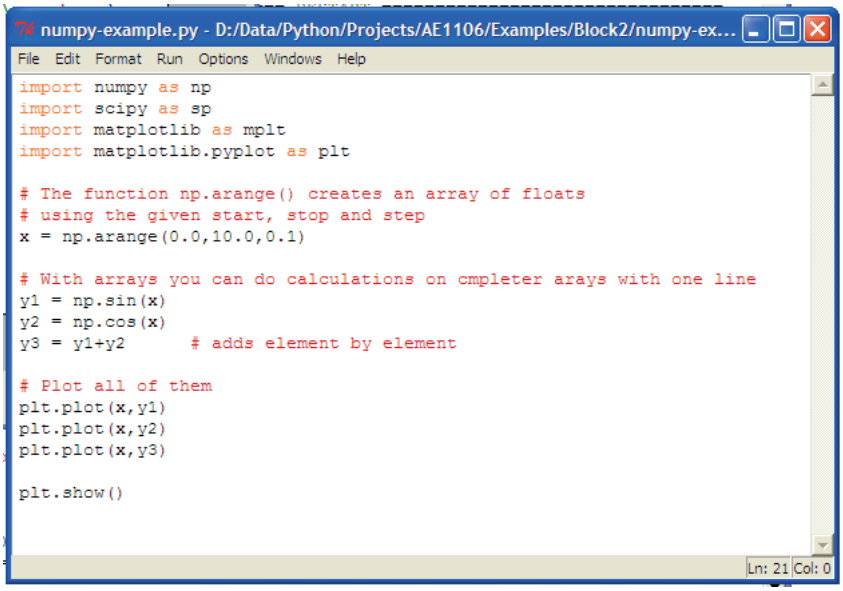
So far we have seen lists and even lists of lists. When we used lists as tables we often kept separate columns in separate one-dimensional lists, so we could select them individually (like xtab, ytab in the previous chapter). And if we used a two dimensional table, we could add small lists of two elements to the lists like this:

```
for i in range(1000):
      x = x + vx*dt
      y = y + vy*dt
      postab.append([x,y])

print("Last x-position: ",postab[-1][0])
print("Last y-position: ",postab[-1][1])
```

But how can we now select only the first column? Unfortunately, postab[:][0] does not give this result. This indicates, as well as many other examples, that two dimensional lists are, as a variable type, despite their versatility, not always the most suitable type for scientific computing and working with large tables in a fast and user-friendly way.

The module Numpy has an advanced list-type which solves this: the array .This type forms the foundations for Numpy and Scipy. With this type you can do computations with entire tables, as easy as if they are one scalar variable.

Look at the example below:



The indexing also has a different syntax although both use the square brackets:

Multi-dimensional lists: `lst[i][j]  a[i][j][k]`
Multi-dimensional Numpy arrays: `arr[i,j] b[i,j,k]`

You can also convert them easily:

Array to list: `lst = list(arr)`
List to array: `arr = np.array(lst)`

Summary of numpy functions to create a numpy array:

np.arange(*start,stop,step***)** Define an array with floats evenly spaced with step (stop not included), difference with range function: it works with floats and the result is a numpy array, Example:
arrange(0.,0.6,0.1) will result in [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]

np.linspace(*start,end,nelem)* Define an array ranging from *start* to, and including, *end* with *nelem* elements, results in a Numpy array, example:
np.linspace(1,3,5) will result in [1.0, 1.5, 2.0, 2.5, 3.0]

np.array(*lst*) Convert list *lst* (or matrix) to array type
np.append(*array,values*) Returns **a copy (!)** of the array with the values appended
np.zeros(*shape)* Create an array of zeros with given shape: e.g. np.zeros((2,3))
np.ones(*shape)* Create an array of ones with given shape: e.g. np.zeros((2,3))

Next to being able to use two indices separated by a comma instead of many pairs of square brackets, there is another useful feature of an array: it is easy to select columns or other parts of the array:

```
xtab = datatab[:,0] # select first column
ytab = datatab[:,1] # select second column
part = datatab[2:3,0:1] # select first two columns of row 3 and 4
```

Numpy arrays can be multiplied by each other or a scalar and used in functions from Numpy (np) like np.sin( ), np.exp(), np.sqrt( ) etc. All functions and operators, work on an element-by-element basis with arrays.

A drawback of arrays is the slower append function. Note the difference in append of lists:

```
import numpy as np

# List append
xtab.append(x)

# Numpy array append
xtab = np.append(xtab,x)
```

The effect is more than a different syntax. The numpy.append function makes a copy of the array with the value appended to it. In this case because xtab is also before the assignment it overwrites the original, so the end effect is the same. The real penalty however is the decrease in

execution speed. With large quantities of data, making a copy takes extra time which can slow down your code immensely. There are two better alternatives, which you sometimes can use:

1. When you know the size, it is better to generate an array of the right size with numpy.zeros(shape) e.g. numpy.zeros(10) or numpy.zeros((4,4)) or a scalar times np.ones(shape) if you need a specific default value.
2. You can use the list-type to append first and then, when it is complete, convert the list to an array.

Exercise 1: *As you have seen in the reader, arrays can make your life a lot easier. Therefore in this first exercise we want you to plot a somewhat harder function, but by making use of arrays, you are much quicker. So make a plot of a hexagon in python, which fits inside a circle with radius 2.*

Exercise 2: *Make a function, when you give a certain n x n array (2D) you get the main diagonal, so e.g. if you have* $A = \begin{bmatrix} 2 & 5 \\ 4 & -1 \end{bmatrix}$ *we want our function to print 2, -1. Make sure this function can work for all n x n matrices.*

Exercise 3: Make a small program that gives the first column of a 2-dimensional list and also make a program that does this for an array. Do you see the difference and how much easier arrays are for this use.

11.3 Logic and Arrays (vectorizing instead of if-then)

There are different ways to use logic in a program which you have vectorized with Numpy. Imagine we want a behaviour comparable with the unvectorized code below:

```
import numpy as np

h      = np.arange(0,20.,1.)
tropo  = 288.15 - 6.5*h
strato = 216.65+0.*h

temp   = []

for i in range(len(h)):
    if h[i]<11:
        temp.append(tropo[i])
    else:
        temp.append(strato[i])
```

There are different ways to do this vectorized:

Using the fact that a True-value behaves a one and a false value as zero:

```
temp = (h<11)*tropo+(h>=11)*strato

swtropo = h<11
```

```
temp = swtropo*tropo+(1-swtropo)*strato
```

Using the conditional indexing to only select the values for which the condition is True and gluing these arrays together with append:

```
temp = np.append(tropo[h<11],strato[h>11])
```

Using the where function to switch between two arrays:

```
temp = np.where(h<11,tropo,strato)   #  where(condition,Truevalue,Falsevalue)
```

Do not confuse this use with the other function of the where-function: when used with only a condition it returns arrays with indices of where it was found to be true.

Exercise 1: *Expand the program so you can calculate the temperature till 32000 m, the temperature gradient between 20 km and 32 km is 1.0 K/km. Make a plot of the temperature against the height.*

Exercise 2*: Plot the discontinuous function f(x) =* $\begin{cases} x^2 \; for \; x \leq 1 \\ 6 - x \; for \; x > 1 \end{cases}$.

Exercise 3*: Extract from the array ([3,4,6,10,24,89,45,43,46,99,100]) with Boolean masking all the numbers:*

- *which are not divisible by 3*
- *which are divisible by 5*
- *which are divisible by 3 and 5*
- *which are divisible by 3 and set them to 42*

11.4 Speeding it up: vectorizing your software with numpy

Check out the two program below. This program uses two different ways to decompose a list of speeds and headings into northern speeds and eastern speeds (assuming no sideslip and no wind).

The example below also times the execution time of both ways.

```
from math import *
import numpy as np
from random import random as rnd
from time import clock

# Generate lists with random speeds & headings
n      = 10000
Vtab   = []
hdgtab = []

for i in range(n):
    Vtab.append(100.+250.*rnd())
    hdgtab.append(360.*rnd())

# Calculating V North and V East the "list way"

vetab = []
vntab = []
```

```
t0 = clock()

for i in range(n):
    vntab.append(Vtab[i]*cos(radians(hdgtab[i])))
    vetab.append(Vtab[i]*sin(radians(hdgtab[i])))

dtlst = clock()-t0
print "Lists took",dtlst,"seconds"

# Convert to arrays
Vtab   = np.array(Vtab)
hdgtab = np.array(hdgtab)

# Calculating V North and V East the "Numpy array way"
t0 = clock()
vntab = Vtab*np.cos(np.radians(hdgtab))
vetab = Vtab*np.sin(np.radians(hdgtab))
dtarr = clock()-t0

print("Arrays took",dtarr,"seconds")

print("So arrays are here",dtlst/dtarr," time faster.")
```

The result show that with 10000 elements the numpy method is about 11 times faster than the list method! And also the numpy code looks cleaner. If we only do it the numpy way, the program can even cleaned up to look like this (without the timing):

```
# import directly, so we do not have to use the prefic np.
from numpy.random import rand
from numpy import sin, cos, radians

n      = 10000
Vtab   = 100.+250.*rand(n)
hdgtab = 360.*rand(n)

vntab  = Vtab*cos(radians(hdgtab))
vetab  = Vtab*sin(radians(hdgtab))
```

The difference between the two approaches is that when you apply a function or an operator on the complete array, or the complete vector, the looping over the elements is handled by low-level code inside numpy. The reason this is faster is twofold:

- the loop takes place in the fast, compiled low-level code of numpy
- list elements can have different types, while all array elements always have the same type, this saves the computer from checking the type for each element and this saves execution time

Changing the code from treating each element to a complete array (or vector) at once, is called vectorizing your code. In general, it is a good rule of thumb that when you can vectorize your code, you should. Sometimes lists are easier or the lists are not that large that it is needed. But for large quantities of data with the same type, vectorizing is nearly always a good idea.

Using two-dimensional arrays and the transpose function, vectorizing can be really powerful for geometrical calculations, see the example below:

```
import numpy as np

# Some random coordinates
n = 4
x = np.array([12, 2,35,11])
y = np.array([ 1,54,23, 7])

# Make two dimensional
x = x.reshape((n,1))
y = y.reshape((n,1))

# Calculate distance matrix
dx = x-x.T
dy = y-y.T
print("dx = ")
print(dx)

dist = np.sqrt(dx*dx+dy*dy)
del dx,dy                          # Free up memory

print("Dist = ")
print(dist)
```

The output of this program is:

```
dx =
[[  0  10 -23   1]
 [-10   0 -33  -9]
 [ 23  33   0  24]
 [ -1   9 -24   0]]
Dist =
[[  0.          53.93514624  31.82766093   6.08276253]
 [ 53.93514624   0.          45.27692569  47.85394446]
 [ 31.82766093  45.27692569   0.          28.8444102 ]
 [  6.08276253  47.85394446  28.8444102    0.        ]]
```

After the reshape x becomes: four rows one column:

```
>>> x
array([[12],
       [ 2],
       [35],
       [11]])
```

x.T means x will be transposed (so rows become columns and the other way around. So in this case x.T is:

```
>>> x.T
array([[12,  2, 35, 11]])
```

Note that this is still a two dimensional array, but with one row (double square brackets).

As you can see in the ouput `x-x.T` results in matrix in which `dx[0,1]` gives `x[0]-x[1]`.

The downside of using vectors this way with a lot of data is that all intermediate answers also become vectors. With large quaitites of data, you can easily consume a lot of memory this way. In that sense, vectorizing sometimes means exchanging speed for memory usage. A good way to avoid running out of memory is therefore delete arrays used for intermediate results. But together with some linear algebra, you can speed things up enormously.

11.5 Matrices and Linear algebra functions

Another very useful type is the matrix. The matrix is a type used within linear algebra. Why don't we use the array type for this? One important reason is that some rules are different when working with entire tables or with matrices.

For instance: A*B will result in entirely different things depending on what type you use. If A and B are arrays, we will get an array where each element is the product of the same elements of A and B. This is called element-by-element multiplication and is what you would expect if you are talking about a table with densities and temperatures for example.

With matrices in linear algebra, we don't want that kind of multiplication, we want a matrix multiplication. This also poses different restrictions on the dimensions. The resulting matrix is a matrix-product of the two given matrices.

There is a special function to do element-by-element multiplication with matrices and there also is a special function to do matrix-multiplication with arrays, but the default * operator uses the most applicable for arrays and matrices as explained above.

So in short: use arrays for tables, like time histories or other scientific computing problems with tables. And use the matrix-type for matrices in linear algebra problems or simulation models with matrices. An example of how to define and use matrices is given below:

```
import numpy as np

A = np.matrix("2 -2 0;1 5 9; 4 2 -5")
print(A)
print(A[1,2]) # row,column as in lin algebra but index starts
              # with 0!

B = A.I
print(B)

C=A*B
print(C)
```

Running this example will first print the matrix A (as a matrix), then a 9, (1=second row, 2=third column). Then it prints the inverse of A, called B here. And finally it should print the identity-matrix to check the inverse calculation.

Note that we use square brackets for indices and 0 as starting index as in Python. But also note how we use a comma instead of two separate pairs of square brackets. Slicing with matrices (and arrays) is much more user friendly and works as you would expect. So `A[:,2]` will give the last column, try for instance:

```
print(A[:,2])
```

There are many ways to define a matrix. The `np.matrix()` or `mat` function accepts lists, arrays and strings as input as in the example above. Using a string with a space between the numbers and a semi-colon between the rows is often the easiest way to define a matrix as in:

```
A = np.mat("2 -2 0;1 5 9; 4 2 -5")
```

With a list this would look as:

```
A = np.mat([[2, -2, 0],[1, 5, 9],[4, 2, -5]])
```

It is possible to convert between arrays and matrices:

```
b = np.mat(a)
c = np.array(b)
```

For some more examples on how to use a matrix, see the following source code. Also note how Scipy also supports all types of Numpy. (So here we use sp instead of np!)

```
from scipy import linalg,mat

eps = 1e-8

# Define a matrix and a vector
A = mat('[1 3 5;2 5 1;2 3 5]')
b = mat('[10;5;3]')

# Calculate determinant to check whether is it solvable
d = linalg.det(A)
if abs(d)<eps:
    print("No solution.")
else:
    print("Determinant = ",d)
    print()

    print("Solve Ax = b with A=")
    print(A)
    print()
    print("en b =")
    print(b)
    print()

    # Solve Ax = b by inverting: B = inv A and a = inv(A)b

    B = linalg.inv(A)
    x = B*b

    print("Solution 1: x =")
    print(x)

    # Solve it in the other way, with the solve function
```

```
print("Solution 2: x=")
print(linalg.solve(A,b))
```

This illustrates some of the functions Numpy provides for matrices: inverse, determinant and solve.

Summarizing the basic matrix functions:

np.matrix( ), np.mat( )	Creates a matrix , argument can be a string like '(1 2;3 4)' or a list, or an array (operators now work with entire matrices)
matrix.**I**	Invert matrix
np.linalg.det(*matrix***)**	Calculates the determinant of a matrix

It is possible to use two-dimensional arrays as vectors and matrices. To use the dot product, it is then required to use the numpy.dot() function.

```
import numpy as np

A = np.array([[2,-2,0],[1,5,9],[4,2,-5]])
print(A)
print(A[1,2]) # row,column as in lin algebra but index starts
              # with 0!

B = np.linalg.inv(A)
print(B)

C = A.dot(B) # alternative: np.dot(A,B)
print(C)
```

Or to solve an equation $A\underline{x} = \underline{b}$:

```
import numpy as np

A  = np.array([[1,0,3],[-2,-1,0],[1,1,2]])
b  = np.array([[1],[-1],[2]])

AI = np.linalg.inv(A)
x  = np.dot(AI,b)

print(x)

# or directly using solve:

x = np.linalg.solve(A,b)
print(x)
```

Note: Unfortunately, lately the Numpy community has been very critical about this matrix type, which is regarded as an unwanted dialect of the array type. Using two-dimensional arrays for matrices means using np.dot(A,B) or A.dot(B) for multiplication and np.linalg.inv(A) for

inverse, you can achieve the same as with the matrix type (but less elegant). This may mean the matrix type may be phased out of Numpy.

Exercises section 11.4

Exercise 1: *Try to solve this linear system A = b, where* $A = \begin{bmatrix} 2 & -5 & 6 \\ 4 & 3 & 8 \\ 5 & -2 & 9 \end{bmatrix}$ *and* $b = \begin{bmatrix} 3 \\ 5 \\ -1 \end{bmatrix}$ *with python.*

Exercise 2: *A motorboat makes a trip of 29.5 km upstream trip on a river against the current in 3 hours and 10 minutes. Returning the same trip with the current takes 2 hours and 6 minutes. Find the speed of the motorboat and the current speed. Try to solve this with a system of equation and python.*

Exercise 3: *Make a linear trend line from the ten points which have coordinates shown below. Use the least squares method to calculate this coefficients for this trend line.*
$x_i = 0.1i$ *where* $i = 1,2,3,...,10$
$y_i = c_1 e^{(-x_i)} + c_2 x_i$ *with* $c_1 = 5$ *and* $c_2 = 2$

11.6 Scipy: a toolbox for scientists and engineers

Numpy forms the foundation of Scipy. It is the basis of the array types and the basic functions such as linear algebra Building on this foundation, there are already a large number of functions available in the modules in numpy. See the list below for an overview of what is inside numpy:

- Array creation routines
- Array manipulation routines
- Indexing routines
- Data type routines
- Input and output
- Fast Fourier Transform (numpy.fft)
- Linear algebra (numpy.linalg)
- Random sampling (numpy.random)
- Sorting and searching
- Logic functions
- Binary operations
- Statistics
- Mathematical functions
- Functional programming
- Polynomials
- Financial functions
- Set routines
- Window functions
- Floating point error handling
- Masked array operations
- Numpy-specific help functions
- Miscellaneous routines
- Test Support (numpy.testing)
- Asserts
- Mathematical functions with automatic domain (numpy.emath)
- Matrix library (numpy.matlib)
- Optionally Scipy-accelerated routines (numpy.dual)
- Numarray compatibility (numpy.numarray)
- Old Numeric compatibility (numpy.oldnumeric)
- C-Types Foreign Function Interface (numpy.ctypeslib)
- String operations

But while Numpy has some useful functions for many applications and even some advanced functions like Fourier transforms (frequency analysis of signals), Scipy contains many more advanced functions, like curve fitting, optimization methods, frequency analysis, statistics:

- Clustering package (scipy.cluster)
- Constants (scipy.constants)
- Fourier transforms (scipy.fftpack)
- Integration and ODEs (scipy.integrate)
- Interpolation (scipy.interpolate)
- Input and output (scipy.io)
- Linear algebra (scipy.linalg)
- Maximum entropy models (scipy.maxentropy)
- Miscellaneous routines (scipy.misc)
- Multi-dimensional image processing (scipy.ndimage)
- Orthogonal distance regression (scipy.odr)
- Optimization and root finding (scipy.optimize)
- Signal processing (scipy.signal)
- Sparse matrices (scipy.sparse)
- Sparse linear algebra (scipy.sparse.linalg)
- Spatial algorithms and data structures (scipy.spatial)
- Distance computations (scipy.spatial.distance)
- Special functions (scipy.special)
- Statistical functions (scipy.stats)
- C/C++ integration (scipy.weave)

On the internet, many more modules, which use Scipy and Numpy can be found. For nearly all fields of science and engineering, modules with many tools are available for free.

In comparison, for a comparable package like MATLAB, often expensive toolboxes are required to use something, which you downloaded from the internet. These toolboxes easily cost thousands of euros, making it a costly decision to try a toolbox. This is, next to the higher versatility and better syntax, one of the reasons why Python+Scipy nowadays is being used instead of MATLAB. IEEE has already called Python+Numpy+Scipy the standard for data analysis. MATLAB will probably become obsolete in the future as it is both more limited and more expensive.

The Python community has the highest amount of PhDs resulting in high quality toolboxes being expanded continuously and growing very fast. For aeronautics, we could still build a better toolbox than the sparse examples which are available.

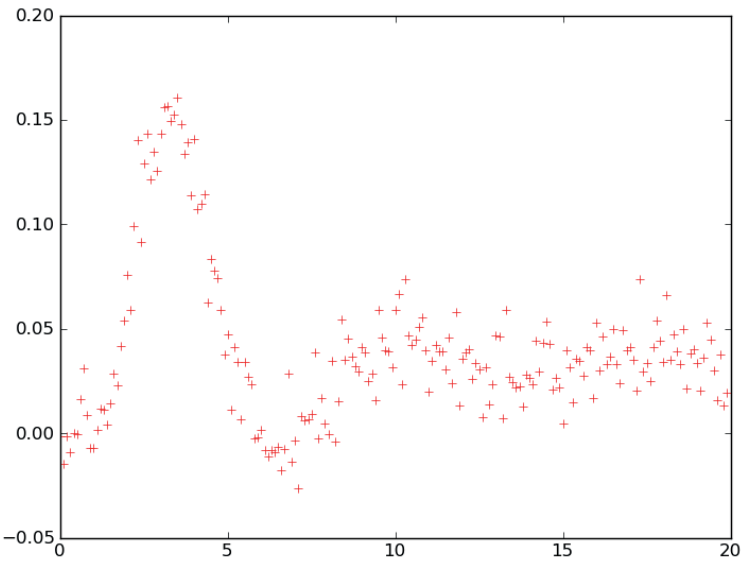
11.7 Scipy example: Polynomial fit on noisy data

When you import Scipy you also get all Numpy functions in Scipy. This means that when you use Scipy the difference between Scipy and Numpy disappears. You can then use the prefix sp instead of np. The following example shows the power of some of the modules in Numpy/Scipy.

Suppose we receive the following data file containing a time history:

```
C
C  Output theta in radians to block response of elevator
C  Elevator = 0.2 radians between 1 and 2 seconds
C  Boeing 737
C  timestamp in seconds
C
0.0 = -0.00122299949023
0.1 = -0.0148544598502
0.2 = -0.00128081998763
0.3 = -0.00912089119957
.........
.........
.........
.........
.........
19.7 = 0.0375150505001
19.8 = 0.0133852241026
19.9 = 0.0195944297302
```

When we plot this data we get this figure, showing a very noisy signal:



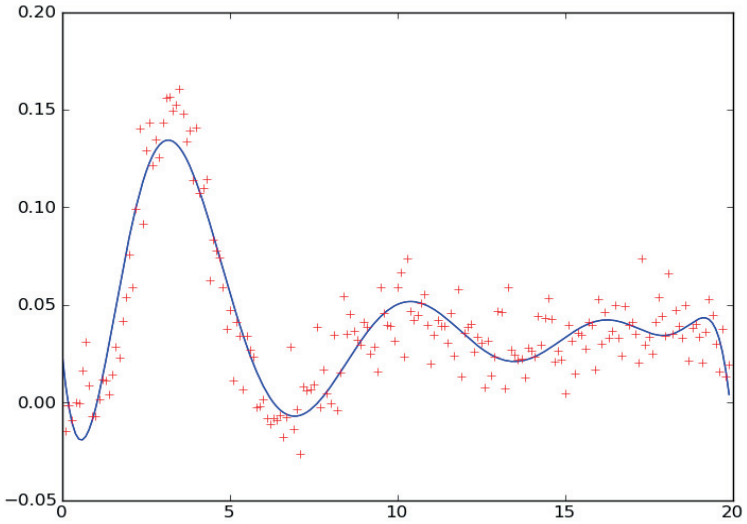
In this plot we can see the original signal, to get this line we can try a polynomial fit, to get a signal more close to the original signal. Note how in this program we use the function `genfromtxt()` from Numpy/Scipy to read the data into a two-dimensional array with one program line! Then we select the two columns.

To fit the data we use the function polyfit, which return the polynomial coefficients, in this case set to a 10^{th} order polynomial. The result is then written to a file.

```
import scipy as sp
import matplotlib.pyplot as plt

# Read file into tables:
table = sp.genfromtxt("flightlog.dat",delimiter="=",comments="C")
xtab=table[:,0]
ytab=table[:,1]

# Polyfitting 10th order

coefficients = sp.polyfit(xtab, ytab, 10)
polynomial = sp.poly1d(coefficients)
ysmooth = sp.polyval(polynomial,xtab)

# Plot

plt.plot(xtab,ysmooth)
plt.plot(xtab,ytab,"r+")
plt.show()

# Write to file

g = open("filtered.log","w")
g.write("C\n")
g.write("C Data smoothed by fitting a 10th order polynomial\n")
g.write("C\n")
```

```
for i in range(len(xtab)):
    line = str(xtab[i])+" = "+str(ysmooth[i])+"\n"
    g.write(line)
    print(line)
g.close()
```

The resulting plot shows both the power and limitation of polynomial fitting:



The curve is indeed a smooth polynomial. The disadvantage is that there are limitations to how a polynomial can fit the data points. Even when a higher order is chosen (10^{th} order seems to be working for a smooth curve with 8 local minima/maxima) it does not give a useful result outside the interval. So while it is very useful for interpolation, it should never be used for extrapolation.

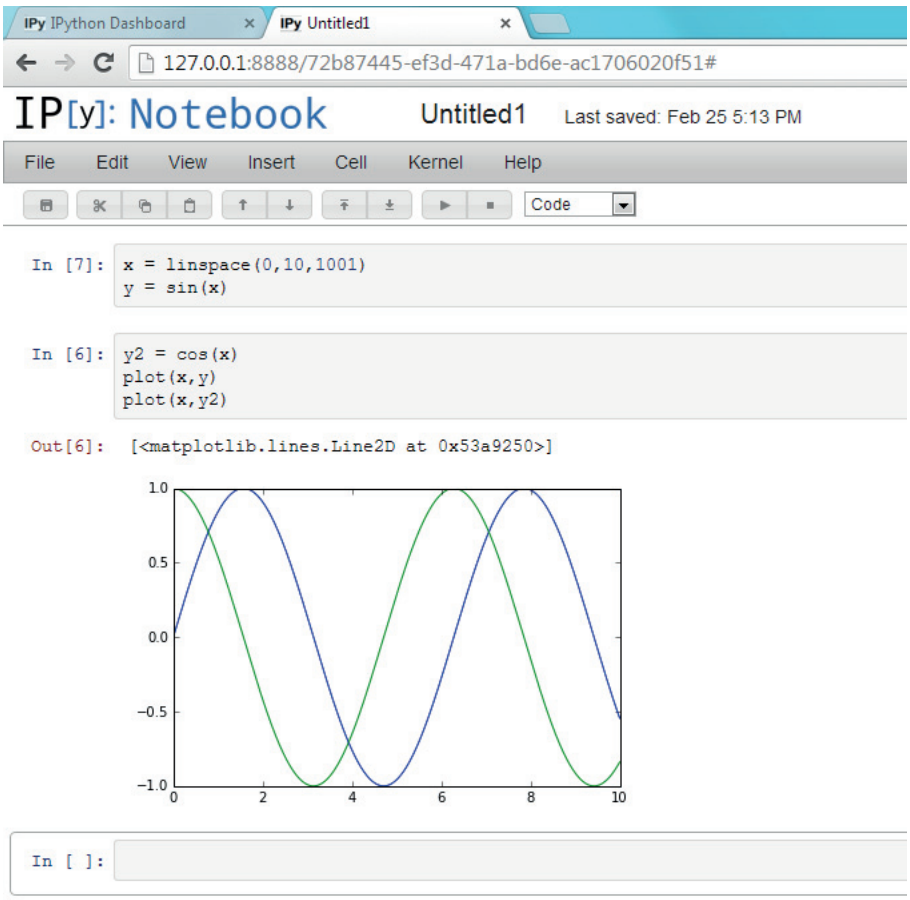
More advanced methods take into account a certain assumed relation of which the parameters will then be estimated. The least squares method, which minimizes the square of the error, can be used for this and is made available by Scipy.

11.7 Jupyter: the iPy Notebook

You can also install jupyter with pip install. With Jupyter you can edit so-called iPython Notebooks. These are a blend between Python and LateX with an excel likt functionality. The result are cells with Python code and the ouptu and documentation is in between the cells. Each cell can run individually, but they all access the same namespace, so one Notebook is like one module. It is accessed in a server-like way, so with the browser.

The iPy Notebook is very convenient user interface for Scientific Computing tasks. It allows you to edit snippets of Python/Numpy code ("Cells") which you run but also can still edit later. You can access all variables between cells, as if you are in one program, but you can also see the output (including plots) in between the code. The result is a blend of a graphical calculator, Python and a spreadsheet program like Excel. You can run an individual cell or all cells and see how the output changes. You can also add cells with raw text to create a living document. Each notebook page can be saved ("downloaded" as it is called in this client-serve set-up) as either a .py Python file or a .pyndb Python notebook file. The Notebook page is both a scratchpad as well as a publication.

An example of how this looks is shown below. As you can see all functions from math, numpy scipy and matplotlib.pyplot have already been imported:



12. Tuples, classes, dictionaries and sets

12.1 New types

Before we continue with Pygame there are two concepts of Python you need to know: Tuples and classes. For classes you only need to know how to use them for now. But a simple example how they are defined helps understanding the principle. In chapter 1 we have seen the list type, which we use regularly in simulations to make tables. For instance like this:

```
import matplotlib.pyplot as plt

# Create tables
ttab = []
ytab = []

# Initialize
y = 100.0
vy =0.0
t = 0.0
dt = 0.01

# Run simulation
while y>0.:
    t = t + dt
    vy = vy -g*dt
    y = y + vy*dt

    ytab.append(y)
    ttab.append(t)

# Plot y
plt.plot(ttab,ytab)
plt.show()
```

Here we note two things: an empty list is created and later elements are added. These elements could be changed later. We could set ytab[i] to a certain value. But what is also notable is the fact that we call the append function (or method) with the variable:

variablename`.append(`*value*`)`

We have seen a similar syntax with strings: *varname*`.sort()` or *varname*`.upper()`. The reason for this is that lists and strings are actually so-called **classes**: a sort of specific variable type with data and functions as a part of its definition.

This syntax is often seen when using modules. Since one of the powers of Python is the number of modules included and freely available on the internet, this deserves some extra attention in the final paragraph on classes.

12.2 Tuples

Lists have been discussed in chapter 2 and in the previous chapter on Numpy we have already seen a new type of lists called arrays, as used by Numpy. In fact, the list-type is the most simple and versatile form of an array- or list-type of variable. It can contain different types and each element can be treated as an independent variable. But there are many more list-like types, one is the so-called tuple.

A tuple is a list but it is immutable (just like strings). This means it cannot be changed once it is created. The variable can be overwritten by a new tuple, but individual element cannot be assigned a different value, nor can elements be added or removed. This is the only difference between lists and arrays. To distinguish between lists and tuples we use the round (normal) brackets to define a tuple. It is also possible to omit the brackets, this also indicates you want to create a tuple. So two valid ways to create a tuple are:

```
origin = (0,0)
pos = 3,4
```

If you call a function with a tuple, you always need the round brackets, see the line with d2 below:

```
d1 = dist(origin,pos)
d2 = dist((3,4),(-3,6))
```

If you would omit the brackets in the second line, Python would think you call the function dist with four arguments.

Tuples can, just like lists, and unlike Numpy arrays, contain a mix of different types such as integers and floats etc..

It seems like a tuple is a list with a limitation, so what are they used for? Tuples can be seen as multi-dimensional values. So for instance if you want to specify an RGB-colour by its red-green-blue components you could use the following assignment to defines these colours for later calls to a graphical library:

```
black   = (0,0,0)
white   = (255,255,255)
brightred = (255,0,0)
red     = (127,0,0)
cyan    = (0,255,255)
```

We've also already seen that `maptlotlib.pyplot.legend()` used a tuple for the legend text, Although this **`legend()`**-function can also be called with a list. In the next chapter about Pygame, tuples are used for colors and positions.

There are many more list-like types. In most cases the list type will work for you. But sometimes a special list-like type can be convenient. If you're curious, check the Python documentation on sets and dictionaries.

12.3 Classes and methods (object oriented programming)

In Python, next to function definitions, you can also define your own variable types and associated functions. This is what classes are. Imagine we could design a new type of variable called Pos, short for position. We want it to hold an x- and y-coordinate and we want to be able to do vector-wise addition. Also a length function will give the length of a two-dimension position vector. Then we would be able to write a program like below:

```
posa = Pos(3,4)
posb = Pos(-1,5)
distvector = posa.sub(posb)
dist = distvector.length()
```

To be able to do this we need to tell Python what our type of variable, our class, is and what the functions should do. This is done by defining a new so-called class Pos:

```
from math import sqrt

class Pos:
    def __init__(self, xcoord, ycoord):
        self.x = xcoord
        self.y = ycoord
        return

    def sub(self,pos2):
        rx = self.x - pos2.x
        ry = self.y - pos2.y
        newp = Pos(rx,ry)
        return newp

    def length(self):
        return sqrt(self.x*self.x+self.y*self.y)
```

After the header a number of **methods** are defined. They use the same syntax as the definition of a function. So a method is a special type of function connected to the class Pos. It is therefore also called by the syntax *varname.methodname(arguments)*.

Note that a special function __init__ is defined first: the so-called “constructor”. It is called automatically upon creation of a new instance of the class (like for posa and posb in the example). This definition of __init__ tells Python how this type can be created and what to do with the arguments that may also be given. In this case, they are simply stored as members x and y. So we note there are two variables stored in a Pos-type of variable: x and y. These are called **members** of the class Pos.

An example of how to use this class (we assume we have saved the above code in the file named CPos.py):

```
from CPos import Pos
# Example of usage:

a = Pos(2,3)
b = Pos(1,2)

c = a.sub(b)

print (c.x,c.y)
print (c.length())
```

You can build your complete program around classes. By first defining your classes including members and methods, then building classes consisting of classes on top of each other, your final program could be very short. This can be done by just calling the highest level of classes like:

```
sim = Sim()
running = sim.start(0.,0.,0.)
while running:
      sim.update(running)
```

This style of programming is called **object-orient programming** (as opposed to normal **procedural** programming) and was for some time very fashionable. It still is, but you also see a return to procedural programming or a mix of the two styles. A disadvantage of object oriented programming is the additional bookkeeping, which you have to do, a huge advantage is the reusability of your classes in different programs. For small to medium sized programs the disadvantages clearly outweigh the advantages. For most scientific computing purposes a procedural program will do fine. However, it could be useful to build your own libraries with specific computations for specific types of physics or technology. For instance a class called aircraft could contain a position, altitude, angles etc.

Overloading operators

It is also possible to define what the operator in a class. For instance the sub function in the class could also be defined in another way, using the two underscore symbols in front of the name and after, a key for the special functions which Python uses, just like __init___ indicates the constructor in Python.

```
def __sub__(self,pos2):
    rx = self.x - pos2.x
    ry = self.y - pos2.y
    newp = Pos(rx,ry)
    return newp
```

The method __sub__ as defined inside this class, will automatically be called when the minus sign operator is used. So the call to sub can then be changed in:

```
c = a - b   # was c = a.sub(b)
```

The following is a list of operators which can be overleoaded (i.e. defined) in a class definition, the name to be used for the method definition is given in the second column.

Operator	Method/function name	Description
+	`__add__(self,other)`	Addition
-	`__sub__(self,other)`	Subtraction
*	`__mul__(self,other)`	Multiplication
/	`__truediv__(self,other)`	Division
%	`__mod__(self,other)`	Modulo/Remainder
<	`__lt__(self,other)`	Less than
<=	`__le__(self,other)`	Less than or equal to
>	`__gt__(self,other)`	Greater than
>=	`__ge__(self,other)`	Greater than or equal to
==	`__eq__(self,other)`	Equal tp
!=	`__ne__(self,other)`	Not equal to
[index]	`__getitem__(self,index)`	Index operator
in	`__contains__(self,value)`	Check contains or not
len	`__len__(self)`	Length, number of elements
str	`__str__(self)`	String representation (for print)

Object oriented programming is beyond the scope of this reader. You do not need to know how to define your own classes. You have already been using classes when calling string methods or list methods. For this course, it is only important that you know that the concept exists and why you sometime call methods or access members. It also explains the syntax of *variablename-period-method(arguments)* like list.append(x).

It is important for you to know how to use the classes and how to call methods inside a class. Especially in Pygame we see two new types of variables being used a lot: a surface and a rectangle. These are classes. One is designed to hold a bitmap, the other to hold a position and a size of any rectangle. In the rectangle class not only methods are used but also members, (like top, width, height) can be assigned a value. This will be shown in examples in the Pygame chapter. Even though the concept of classes and object-oriented programming may be difficult, using modules with classes is surprisingly easy and user-friendly, as we have already seen with lists and strings and we will also see in the Pygame chapter.

12.4 Dictionaries & Sets

Dictionaries
Two special types of lists can be handy. Dictionaries are lists where you do not use an integer index but a key to look up a value.

An example of a **dictionary**:

```
>>> ages = { "Bob": 20 , "Alice": 18 ,"Jacco": 29 }
>>> ages["Jacco"]
29
>>> ages["Jet"]

Traceback (most recent call last):
  File "<pyshell#10>", line 1
    ages["Jet"]
KeyError: 'Jet'
>>>
```

This error can be prevented by first checking whether 'Jet" is in the keys of this dictionary, using the keys function:

```
if "Jet" in ages.keys():
      print ages["Jet"]
```

Sets
Sets are lists used for unordered collections of unique elements. It is used to check membership of a collection, overlaps of collections, etc.

It is defined similarly to a list (square brackets) with the function set. Example of the use of **sets**:

```
>>> a = set([3,1,34,65,2,2,1])
>>> a
set([1, 34, 3, 2, 65])
>>> if 2 in a:
      print 'yes'

yes
>>> 2 in a
True
>>> b = set([1,6,7,10,2,3,7])
>>> a|b  # which elements are in a or b?
set([1, 34, 3, 6, 65, 10, 7, 2])
>>> a&b  # which elements are in a and b?
set([1, 2, 3])
>>>
```

The difference between a set and a list, is that in a set an element does not have specific place and each element is only in once. It is a collection as used in the mathematical set theory: an element is either a part of a set or not.

Imagine you have two lists with a set of numbers, and you simply want to know whether they use the same digits, than converting them to sets and comparing it will do this for you.

There are a number of methods which can be used on sets:

```
a = set([1,2,5,3,6,7,0])

b = a.copy() # returns contents of a as a set (shallow copy)
b.add(100)   # adds 100 to set b (but not to set a)
xdif = b.difference(a) # return elements in b which are not in a
overlap = a.intersection(b) # return set items which are in a and in b
together = a.union(b) # return a set with elelemtns in a or in b
print(a.issubset(b)) # Will print True, cause a is subset of b
```

For a complete overview, including the operators, type "**help(set)**" in the Python shell.

13. Pygame: animation, visualization and controls

13.1 Pygame module

The Pygame module contains a set of user-friendly modules for building arcade-like games in Python. But because it has several drawing routines as well as key controls, it is also a very convenient library to make any 2D graphics, especially moving graphics with some key controls. Therefore it has also become the default graphics library. It is often used for other purposes than games, like animation, visualization of simulation and all other non-moving drawings which need something more versatile than Matplotlib, which produces graphs but is not a graphics library. Also key and mouse controls are part of Pygame and can be added to pan, zoom or control a simulation.

In this chapter we will explore different modules inside Pygame like display, event, key, mouse, image, transform and draw, as well as the new types (classes to be exact): surface and rect. There are many more modules inside Pygame for you to explore like, music, joystick, etc. but these will not be discussed here. Next to Pygame, some basics of setting up a game or simulation will be discussed such as the 'game-loop' and timing issues.

Just as with Numpy and Matplotlib, Pygame is a third party add-on. So before you can use Pygame, you need to add the import line. Also you should add two calls to Pygame at the beginning and end of your program for initialization and cleaning up, resulting in the following three mandatory lines:

```
import pygame
pygame.init()

....
....
(your program will be in between these calls)
....
....

pygame.quit()
```

The init-call has no immediate visible effect, but it is required to avoid having to initialize each module of Pygame independently. So calling **`pygame.display.init`**, **`pygame.key.init`** is not necessary when you include this one call.

`pygame.quit()` will have a visible effect: it will close any Pygame windows still open. If during the development your program crashes due to an error and the Pygame window is still open, type **`pygame.quit()`** in the shell to close it. (or **`pg.quit()`** if you used **`import pygame as pg`**)

Pygame has a very good online reference manual at http://pygame.org/docs . In the top section of this page, you see the name of each module inside Pygame. Click on these names to get a list of functions in that module, and click on the function name in this list to get a full description.

13.2 Setting up a window

Before we draw anything, we need a window to draw in. The way to do this is using a function in the display module, which is called `pygame.display.set_mode()`. The various possibilities of this function are explained in the documentation of Pygame. To get an impression, here is an example call which creates a window of 600 pixels wide and 500 pixels high:

```
reso = (600,500)
screen = pygame.display.set_mode(reso)
```

Or in a similar way (mind the double brackets, the resolution is a so-called tuple, see section 12.2):

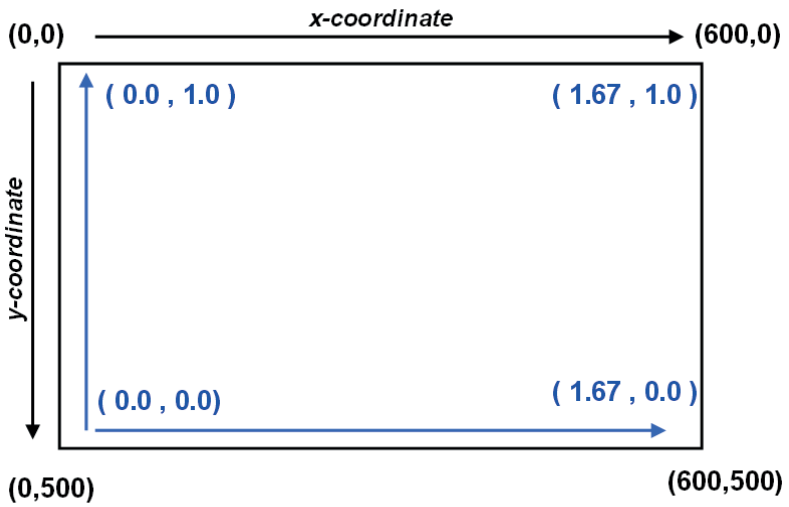
```
screen = pygame.display.set_mode((600,500))
```

The function `pygame.display.set_mode()` returns a surface, which is stored here in the variable name screen (could have been any name). In this case the surface refers to the video memory of the screen, so we have called this variable screen in our example but it could have been any other name like win, window1, scr, etc. In our example `screen` is now a surface we can draw on.

In computer graphics, a coordinate system different from mathematics is used, mainly for historical reasons. The top left corner is the origin and the y-coordinate runs from zero in the top to the bottom, so in our example y=500 pixels indicates the bottom line. X-coordinates are from left to right. This stems from the text terminal which had line zero and column zero in the top left. So the window, which we just created, has the following coordinate system:

In the computer graphics world things get really complicated in three dimensions. In the graphics coordinate system, the Z-axis points positive to the viewer, in effect creating a left-handed (!) axes reference system, where cross products and other functions work just the other way around.

So screen coordinates work in a non-standard way and are also always integers. For these two reasons, it is standard procedure to define your own so-called world coordinates (with a right-handed reference frame), which you use for your model. And when necessary you convert them to screen coordinates, normally just before we need to plot, draw or blit. It also allows us to change the window size later (e.g. to full screen), without changing our model. An example of a simple 2D world coordinate system is:



In this case we can do calculations using any position in floats, like do numerical integration in floats and then only when we draw convert our float world coordinates to the integer screen coordinates just before plotting with the following lines of code. In the example below the world coordinates are (x,y) and are converted to screen coordinates (xs,ys):

```
xmax = 600
ymax = 500
reso = (xmax,ymax)
screen = pygame.display.set_mode(reso)

….
…..
      x = x + vx*dt
      y = y + vy*dt
      xs = int(x/1.67*xmax)
      ys = ymax-int(y*ymax)
```

13.3 Surfaces and Rectangles

There are two new concepts to grasp in Pygame: they are called Surface and Rect. They each have their own section in the http://pygame.org/docs documentation. A surface is a type to hold an image, a bitmap. It can be an entire screen, an image loaded from a file or (often) a smaller part. You can draw on a surface; you can copy and paste any surface, or parts of it, on another surface. A surface can also have a transparent background so that when you paste it over another surface some pixels will keep the color of the original surface.

All in all, it is a very versatile type allowing you to manipulate or transform (scale move, copy/paste, rotate) bitmaps.

When you paste one surface onto another, this is called blitting in the computer graphics world. It stems from the "block of bits" of the video memory which is transferred with one call to a blit function. Before we can do this we also need to specify where we want this rectangular image to be pasted.

When we have loaded an image from the disk, we need to know how large it is and where we should be able to position it onto another surface later. This is where the rectangle class comes in. A rectangle is not the actual surface but it contains some parameters: the size and position of a surface. It has several members, which we can assign values to. The neat thing is that we do not need to worry about the bookkeeping: when you change one value, the other ones will be changed automatically when needed. One simple example is given below. To clear the screen, we draw a black rectangle. For this we need to specify the position and scale first. If we get the rectangle from a surface to measure the size, the position is by default set to (0,0) for the top left corner. To do this, we use the method `get_rect` in the Surface class (see the section on Surface in the Pygame documentation for a full description of the methods of surface).

```
black =(0,0,0)
scrrect = screen.get_rect()
pygame.draw.rect(screen,black,scrrect)
```

A Rect has the following members, which you can read or assign a value to:

```
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

Center is a tuple equal to (centerx,centery). The user can choose to position the rectangle using any combination of these members.

Another example where the Rect class is used is given below, where an image is loaded from a file on the hard disk. Generally, we do this loading from a file only once at the beginning of our program, because accessing the hard disk has a huge execution speed penalty. The example shows how to use the rectangle: first we get the size from the surface object and then we position it and use it to blit the surface on the screen surface:

```
ship       = pygame.image.load("rocket.gif")
shiprect   = ship.get_rect()
…..
…..
while running:
      shipx = shipx + vx*dt
      shipy = shipy + vy*dt

      shiprect.centerx = shipx
      shiprect.centery = shipy

      screen.blit(ship,shiprect)
```

In the code we can see how the rectangle is used to position the bitmap, which was read from rocket.gif, on the screen. The same call can be used with a tuple containing the coordinates of the top-left corner of the surface. As visible in the syntax, blit is a method from the Surface class, hence it is called as a method, so with a period behind the destination surface (=variable named "screen" in our example).

13.4 Bitmaps and images

In the previous paragraphs we have seen that surfaces allow you to load bitmaps from the disk and blit them to the screen with the following two functions:

```
scr      = pygame.display.set_mode((500,500))
ship     = pygame.image.load('lander.gif')
shiprect = ship.get_rect()
      .........
       .........
      shiprect.center = (xs,ys)
      scr.blit(ship,shiprect)
```

A few remarks about using these functions:

As said before, it is important, to load the bitmaps before the actual game loop. Accessing files, to load an image, in general takes a lot of time. This means it might cause hick-ups or delays in your program, if you do this during the loop.

Sometimes bitmaps do need some editing before they can be used. When blitting images on a surface a transparent background is often required. This can be edited with a painting program such as Paint.net, Gimp, Paint Shop, Corel Draw or Photo shop and deleting the background so it becomes the transparent colour. Save the file in the GIF format (or PNG) as these formats allow transparent backgrounds. Also use these programs to change colors, size or perform rotations.

If you need one bitmap in a lot of different orientations, editing it with a paint program can be cumbersome. These operations can also be done much easier with pygame in your program. This

can be done using the transform module which allows you to manipulate bitmaps/surfaces. Some examples of functions available in `pygame.transform`:

pygame.transform.flip(*surface,xswith,yswitch*) flip vertically and horizontally, returns new surface

pygame.transform.scale(*surface,(newwidth,newheight)*) - resize to new resolution, returns new surface

pygame.transform.rotate(*surface,angledeg)* rotate an image, angle is float in degrees, rotate an image, returns new surface

pygame.transform.rotozoom(*surface, angle, scale*)- filtered scale and rotation, angle float in degrees, scale factor also float, returns new surface

pygame.transform.scale2x(*surface*) specialized image doubler, returns new surface

pygame.transform.smoothscale(*surface,(newwidth,newheight*)) - scale a surface to an arbitrary size smoothly, returns new surface

Be aware that you do not scale or rotate inside the game loop unless it is absolutely necessary. In most cases it pays off to generate different surfaces for all possible orientations beforehand, store them in a list and just use the appropriate one during the game loop by setting the index with the angle rounded off to 45, 20 or 5 degrees. The same goes for scaling. Transforming bitmaps is rather computational intensive and in general it is the goal to do any time consuming operation as much as possible before the actual running of the game loop.

13.5 Drawing shapes and lines

The draw module contains functions to draw lines and filled or unfilled shapes such as rectangles, circles, ellipses and polygons:

pygame.draw.rect	- draw a rectangle shape
pygame.draw.polygon	- draw a shape with any number of sides
pygame.draw.circle	- draw a circle around a point
pygame.draw.ellipse	- draw a round shape inside a rectangle
pygame.draw.arc	- draw a partial section of an ellipse
pygame.draw.line	- draw a straight line segment
pygame.draw.aaline	- draw fine anti-aliased lines

A few notes on using these functions:

The coordinate system used is explained in the section on setting up your screen. It runs from (0,0) in the top left corner to the maximum x and y in the bottom right. All sizes and positions are integers.

Colours are specified with a tuple (red,green,blue), three numbers each ranging from 0 – 255 to indicate the amount of each colour channel present in the mixed color. For readability it helps to define a few colours at the beginning of your program and use these in the calls:

```
black = (0,0,0)
cyan = (0,255,255)
white = (255,255,255)
background = (0,0,63)
foreground = white
```

Some draw functions allow to switch on what is called “anti-aliasing”. This means a pixel on the edge of the line will get a colour in between the foreground colour and background colour depending on the ‘amount of line’ which is in the pixel. This avoids the jagged lines you will get with normal bitmapped shapes. One disadvantage of this is that it takes more time and memory: the shape is first drawn in a higher resolution and then “anti-aliased”. Another disadvantage is that you later on cannot select the pixels which are part of the background and not a part of the shape. It depends on your application whether you want to use anti-aliasing.

13.6 When our drawing is ready: pygame.display.flip()

Animation consists of nothing else than redrawing every frame over and over in a simulation or game. To avoid flickering images when the screen is cleared for the next drawing, this is first done in a part of the video memory which is not visible. There we clear the screen, and either with drawing (rectangles, circles and lines) or blitting the next image is created. Once it is finished, we can show it on the screen. This also means we will not see anything until we copy this video memory to the screen. This is done with the following call, to be used at end of all your draw and blit calls in the loop.

```
pygame.display.flip()
```

If you have used the default settings when creating the window with `pygame.display.set_mode()`, a call to flip updates the window with the video memory. When you use the option double buffering, you have two areas of video memory, so two surfaces which will be used simultaneously: one will be shown on screen, the other you can draw on. Once you’re done drawing the frame, you swap the function of both surfaces. The advantage is that you do not need to start with an empty frame then, but can use the one-but-last frame as a start. In practice, to take advantage of this you might need a lot of bookkeeping but it might result in a higher execution speed.

Using the full-screen option is often only done when the game or simulation is ready, debugged and tested, since debugging is severely hampered by a crashing full screen application!

13.7 Timing and the game-loop

Imagine we want to let a rocket take off on the screen. We could use the following code to do so:

```
import pygame
pygame.init()

reso = (600,500)
screen = pygame.display.set_mode(reso)
scrrect = screen.get_rect()

black =(0,0,0)

ship       = pygame.image.load("rocket.jpg")
shiprect   = ship.get_rect()
shiprect.centerx = 250

for y in range(500,-100,-2):
    shiprect.centery = y
    pygame.draw.rect(screen,black,scrrect)
    screen.blit(ship,shiprect)
    pygame.display.flip()

pygame.quit()
```

Now when we run this, it could turn out that our ship moves too fast or too slow. To fix this, we will then adjust the third argument of the range function in the for-loop, currently set to -2. However, on another computer the speed would again be different. Also, when another application in the background needs the CPU for a short period, our rocket will hamper before continuing. This is because we have no control over the real speed of the rocket. For this we need more than controlling the position, we need to control the timing in our game as well. Or at least measure it and calculate the elapsed time (time step) since we last drew our ship, so that we can calculate the required new position with the speed and right time step based on the elapsed time since the last frame.

There are two principles we can use to make sure our simulated time runs in accordance with the real time:

Method I: Fixed time step
We use a constant time step similar to our previous numerical integration examples. We check whether the real time is equal to or larger than our next to be simulated time, if so, we make a time step dt.

This is how this could look when coded in Python:

```
import pygame
# initialize clock
pygame.init()
tsim  = 0.0
tstart =  0.001*pygame.time.get_ticks()
dt = 0.1
……
……
running = True
while running:
      trun = 0.001*pygame.time.get_ticks() - tstart
      if trun+dt >= tsim:
            tsim = tsim + dt
            vx = vx+ax*dt
            vy = vy+ay*dt
            x = x + vx*dt
            y = y + vy*dt
            ……
            ……
```

The advantage of this method is that you know what the time step is. This also allows for more advanced numerical integration methods and guarantees the stability of your simulation. When using a variable time step (Method II) there is a risk of getting a too large time step, which can result in overshoots and even a runaway of your simulation. The disadvantage of this method, that it requires a time step, which is large enough to make sure the simulation can always keep up with the real time. If you make this time step too small, the simulation may lag behind or run at a variable speed, resulting in quirky speeds and movements

Method II: Variable time step
In this case we simply measure the time elapsed since the last time we update the simulation. This difference is our time step dt, which we then use to update everything including the time. It looks much simpler, and when the computer is fast enough it will also use this for higher update frequencies. The downside is that the reverse is also true: if the computer is too slow or occasionally too slow, the simulation might cause problems because of a very large dt. It is possible to safeguard this and catch up later, but then the code gets more complex than we want to use for now. (It becomes basically a mix of the two methods.)

This is how the code looks in Python when using the method of the variable time step. The variable t0 here keeps the previous time when everything was updated:

```
import pygame
# initialize clock
pygame.init()
t0  = 0.001*pygame.time.get_ticks()
……
……
maxdt = 0.5       # time step limit to avoid jumps
running = True
while running:
      t = 0.001*pygame.time.get_ticks()
      dt = min(t-t0,maxdt)    # set maximum limit to dt
      if dt>0.:
            t0 = t
            vx = vx+ax*dt
            vy = vy+ay*dt
            x = x + vx*dt
            y = y + vy*dt
            ……
            ……
```

One of these two mechanisms forms the basis of our loop which is executed while running. This is generally called the game-loop. In principle, it runs indefinitely until a quit event is triggered. This quit event can be multiple things:

- Escape key is pressed
- Some conditions in our simulation are met (simulation ready in simulation, player dead in games, exception has occurred, boss enemy detected, etc)
- Quit event is given by windows (so the user has clicked the cross of the window).

In our example so far, just setting running to False will make sure the loop ends:

```
running = True
while running:
    .....
    .....
    if y<0.:
        running = False
    .....
```

Another way to achieve this, is using the break command (which I personally see as less elegant):

```
while True:
    .....
    .....
    if y<0.:
        break
    .....
```

In general within a game loop we see the following elements:

- check for time step
- get input from keyboard, mouse or any other source
- update model with numerical integration
- draw new frame
- check for quit events

Before the game loop our model is initialized, graphics are loaded and set-up and our simulation is initialized. We have seen how to control time, how to numerically integrate and how to draw a frame. But we still need to know how to process input from keyboard and/or mouse.

13.8 Input: keyboard, mouse and events

When running a simulation or game, you may want the user to control certain aspects of the simulation. This could be triggering some controls, events or control display functions such as zoom and pan. Also the user might control when the simulation starts or stops. Until now we have used the input function to get user input. This does not fit this purpose: the complete program stops, we need to go to the Python shell window and we do not want the user to press enter every time step. So we need a way to check the state of the keys while we keep running. If possible we should also be able to check two keys being pressed simultaneously. Using the pygame.key module this is indeed possible. It can be achieved by including the following two lines:

```
pygame.event.pump()
keys = pygame.key.get_pressed()
```

The first line is required because of the way windows handles events, like keys being pressed. The second line collects all key states in a long list of logicals. Each key has a fixed position in this list of logicals. When this logical is True, the key is currently held down and when the key is not pressed, the logical is False.

So imagine we want to check for the Escape key, how do we know which logical to check? Pygame has a list of variables (integers) with indices for every key (check http://pygame.org/docs/ref/key.html) that you can use. For example to check the Escape key we use (after the storing the logical in our variable **`keys`** with the above piece of code):

```
if keys[pygame.K_ESCAPE]:
      running = False
```

Some examples for other indices we can use (always with pygame. in front of this name if you've use `import pygame`):

`K_0`	0 - key
`K_1`	1 - key
…..	
`K_9`	9-key
`K_a`	A-key
`K_b`	B- key
….	
`K_z`	Z-key
`K_SPACE`	Space bar
`K_UP`	Up arrow key
`K_DOWN`	Down arrow key

`K_LEFT`	Left arrow key
`K_RIGHT`	Right arrow key
`K_F1`	F1 function key
`K_TAB`	Tab-key
`K_DELETE`	Del-key
`K_RSHIFT`	Right shift key
`K_LSHIFT`	Left shift key
`K_LCTRL`	Left Control key
`K_RCTRL`	Right Control key
`K_LALT`	Left Alt key
`K_RALT`	Right Alt key

Similarly we can get the state of the mouse buttons with `pygame.mouse.get_pressed()`, which returns three logicals for the three mouse buttons. The function `pygame.mouse.get_pos()` returns the current position of the mouse.

Both key- and mouse-functions will return only the useful values if your pygame-window is running in the foreground (or in windows speak: has focus).

Another more advanced way to handle the mouse is to use the windows event handler. For instance for the quit-event (somebody tries to close the window, e.g. by clicking on the red button). An example of how this event handling could be used is given in the following piece of code:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        running = False
    elif event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
        running = False
    elif event.type == pygame.MOUSEBUTTONDOWN:
        mx,my = event.pos

        ......
    elif event.type == pygame.MOUSEBUTTONUP:
         ......
```

Many of these events contain data, e.g.on where the mouse was clicked, which button or which key was pressed. See the example above for using the mouse position. Other names of members of the different event type are listed below:

Event type: pygame.	*Contains:*
QUIT	**none**
ACTIVEEVENT	**gain,state**
KEYDOWN	**unicode,key,mod**
KEYUP	**key,mod**
MOUSEMOTION	**pos,rel,buttons**
MOUSEBUTTONUP	**pos,button**
MOUSEBUTTONDOWN	**pos,button**
JOYAXISMOTION	**joy, axis, value**
JOYBALLMOTION	**joy, ball, rel**
JOYHATMOTION	**joy, hat, value**
JOYBUTTONUP	**joy, button**
JOYBUTTONDOWN	**joy, button**
VIDEORESIZE	**size, w, h**
VIDEOEXPOSE	**none**
USEREVENT	**code**

Most of the times the event handling is not required, but as a minimum handling the quit event is considered good practice: it allows the user to close the window.
Note: it is important to always include the event pump. If not the Pygame window will say “Not responding” as the windows OS events will not be processed.

13.9 Overview of basic pygame functions

import pygame Below it is assumed, you have imported Pygame simply as pygame

Two new types
:
Surface*: A surface is an image or a part of an image (can be small bitmap) in memory*
Rect*: A rectangle holds only a position and size of a rectangle as members*

pygame.init() Initialize all Pygame modules
pygame.quit() Closes all Pygame windows and quits all modules

pygame.display.set_mode((*xpixels*,*ypixels*)) Create a window with specified resolution in pixels, returns a ***surface*** with the video memory of the screen. Resolution is a so-called tuple, set as (xmax,ymax).

pygame.draw.rect(*surface,colour,rect*) Draw filled rectangle in *surface*, *colour* is tuple (red,green,blue) with values <= 255, position in *rect*

pygame.display.flip() Update the screen with the video memory (so display screen surface)

pygame.time.get_ticks() Returns system time in millisecond since pygame.init()

pygame.image.load(*filename*) Load an image from a file (a.o. jpg,gif,bmp,png, etc.) returns a ***surface*** with the image

surface.get_rect() get the size of a surface in a ***rectangle*** (**top** and **left** will be zero)

surface*.blit(*sourcesurface,rect) Pastes the source surface on the *surface* at the position specified in ***rectangle rect***

Moving a rectangle can be done in different ways, by setting members of **rectangle**.
rect*.centerx** = 456 ***rect.top = 0
rect*.centery** = 240 ***rect.left = 0

pygame.event.pump() Flush event queue (to avoid hanging app, and to poll keyboard)

var = **pygame.key.get_pressed**( ) Poll the keyboard: logical array for all keys, True if pressed
if *var*[pygame.K_ESCAPE] : How to test for Escape key: test the logical array with the right index (see documentation for other key indices)

pygame.transform module contains operation on surfaces like rotate & scale.

pygame.mixer.music contains functions to load, play and save sound files

More information on these and many other pygame functions can be found in the on-line reference guide: http://pygame.org/docs

Exercise 1: *With the knowledge you achieve each section, we want you to make a relatively easy game. So the first exercise is to make a screen with a resolution of 750x500 px, besides this it should also have a blue background.*

Exercise 2: *Place an image of an airplane at the left side of this screen. Google for a nice plane "plane no background", flying to the right*

Exercise 3: *Now we want to include a time loop so that your airplane flies straight forward with constant speed over your screen.*

Exercise 4: *The final step to make your game actually fun is to add key input, so make sure your plane can move up and down when you press the arrow up and when you press the arrow key down. Also make the escape key such that if you press it you quit the game.*

14. Exception Handling in Python

A powerful feature of Python is that it allows you to trap runtime errors using the TRY and EXCEPT statements. Be aware that this is also a way to obscure errors and can quickly make your program impossible to debug. In general it is better to prevent errors than to catch them.

Still, it can sometimes be a very useful feature, so we show a few examples here of how to use try and except.

The first example of an a,b,c formula solver for second order polynomial could check for a negative D by simply catching the error of a negative square root:

```
import math

print("To solve ax2 + bx + c = 0 ,")

a = float(input("Enter the value of a:"))
b = float(input("Enter the value of b:"))
c = float(input("Enter the value of c:"))

D = b**2 - 4.*a*c

try:
    x1 = (-b - math.sqrt(D)) / (2.*a)
    x2 = (-b + math.sqrt(D)) / (2.*a)
    print("x1 =",x1)
    print("x2 =",x2)
except:
    print("This equation has no solutions.")
```

To make it even more advanced: it is possible to check which error was caught:

```
import math

print("To solve ax2 + bx + c = 0 ,")

a = float(input("Enter the value of a:"))
b = float(input("Enter the value of b:"))
c = float(input("Enter the value of c:"))

D = b**2 - 4.*a*c

try:
     x1 = (-b - math.sqrt(D)) / (2.*a)
     x2 = (-b + math.sqrt(D)) / (2.*a)
     print("x1 =",x1)
     print("x2 =",x2)
```

```
except ZeroDivisionError:
      print("this is a first order equation: a=0")
      x = -c/b
      print("Solution x =",x)

except:
      print("This equation has no solutions.")
```

Just like other flow control statements, try statements can be nested:

```
try:
    x1 = (-b - math.sqrt(D)) / (2.*a)
    x2 = (-b + math.sqrt(D)) / (2.*a)
    print("x1 =",x1)
    print("x2 =",x2)
except ZeroDivisionError:
    print("this is a first order equation: a=0")
    try:
        x= -c/b
        print("Solution x =",x)
    except:
        print("No x found.")
except:
    print("This equation has no solutions.")
```

Examples of types of errors you can catch (the names speak for themselves):

```
ZeroDivisionError,    OverflowError,   NameError,
IndexError,           TypeError,       KeyboardInterrupt,
SyntaxError,          SystemExit,      FloatingPointError,
KeyError,             ValueError,      ......
```

When an error occurs, the name of the error is always given in the shell. A complete list can also be found in the Python (the documentation provided with Python) under the header Exceptions.

15. Distributing your Python programs using Py2exe

14.1 Making an .exe of your program

Many students have asked me how they can share their Python programs with people who haven't got Python installed on their computer, for example by converting your Python program into an executable. There is a tool for this, called py2exe. Py2exe is included in Python(x,y). At first glance using this tool may seem quite complex. So to show how you can use py2exe to make an executable of your game, I'll describe how I did this with an example game that I've made with Pygame. Here is some info on the game first:

Example: Mazeman game

It is a game written in Python (2.6) using the third-party module Pygame and the Tkinter module, which is included in Python. It is a variation on Pac-man, but it allows the user to edit simple text files containing the maze lay-out including an unlimited number of ghosts (and a player starting position). Many creative mazes are possible, just right-click Edit in the File Open menu to create and save your own. The program reads the file the user selects and starts the game with the maze as specified in this file. The program scales all graphics to a window size, which covers max 80% of the screen. The sprites are non-animated, continuous movement.

The game has two python files: mazeman.py and toolsmaze.py, the latter of which all functions are imported in mazeman.py (with the "star"-import). The are three subfolders with data files named bitmaps, sounds and choose-your-maze. The program uses fonts to display the score text on the top line. The main module, with which the program is started, is called mazeman.py.

Install py2exe

To install py2exe, simply use pip install in a Command prompt/Console with Admin rights (preceded by "sudo" on Mac and Linux) :

```
pip install py2exe
```

or download it yourself:

https://pypi.python.org/pypi/py2exe/0.9.2.0#downloads

Convert your Python program to an executable

Using the py2exe tutorial, the next step is two make two files with IDLE (or Notepad), in the same folder as where our game modules mazeman.py and toolsmaze.py are located.

- setup.py
- runsetup.bat

First I also tried the pygame2exe script provided at pygame.org, but this kept giving me errors and I also did not fully understand what it was trying to do. In the end it turned out, I did not need all these bells and whistles, as long as I was willing to add my data files (bitmaps etc.) and some DLL-files manually later, as we will see.

Then we make our own python script called setup.py in the same folder of our Python game. It contains only the following few lines:

```
from distutils.core import setup
import py2exe

setup(console=['mazeman.py'])
```

You can change the word 'console' into 'window' if you're not using the console window (so you don't use print( ) or input( ). In the final line, we only mention mazeman.py and not toolsmaze.py, since this is the main module. Py2exe will read the import-statements in there and then automatically also add toolsmaze.py, as well as Pygame and Tkinter (and os and sys and any other modules we use)

We need to run this script outside IDLE. For this we use the following file, a so-called batch file (extension .bat), which we call runsetup.bat (make textfile with Notepad, which you rename to runsetup.bat):

In this runsetup.bat, you edit:

```
python setup.py py2exe
pause
```

This batch file starts our setup.py The pause command at the end ensures that we have the time to read potential error messages if it crashes, before the console window closes again. If there are fatal errors, we need to fix those errors, sometimes leading to tidying up your code a bit.

If it runs successfully, it will create a subfolder called `dist` containing your program (`mazeman.exe` in our example) and a lot of other files needed by the executable. Still we also need to add some files manually before we're ready.

Adding additional files to make your program run

First of all we need to copy our data files to this dist folder, so our mazeman.exe will find them. In our example, the three subfolders named: bitmaps, sounds and choose-your-maze.

In case you need additonla compiler for windows, get them here:

https://wiki.python.org/moin/WindowsCompilers

Next we go to the pygame subfolder in your python folder:

```
C:\Python26\Lib\site-packages\pygame
```

Py2exe should have copied the relevant DLLs, but regularly misses out on a few (often the Font ones). So from this directory we copy (again be careful not to delete or move them) all DLLs (so

sort by type and select all of type “Application Extension”) to your dist folder. We do the same with the font file: `freesansbold.ttf`.

In short we copy these files from the pygame folder to our dist subfolder:

```
*.dll
*.ttf
```

To see whether our program needs any additional files, it may be useful to start our program mazeman.exe with a batch files with a pause command, so we can read warnings and error messages in the console. Let’s call this `runme.bat` for example (in the dist folder as well) . Create it with notepad or idle):

```
mazeman.exe
pause
```

Then run it to test it and see if you need to add any other files. Sometimes you may need other DLLs from third party module you use. You can find these folders in your Python directory (in **`C:\Program Files\Python36\Lib\site-packages\`**), and basically do the same as we did above for pygame.

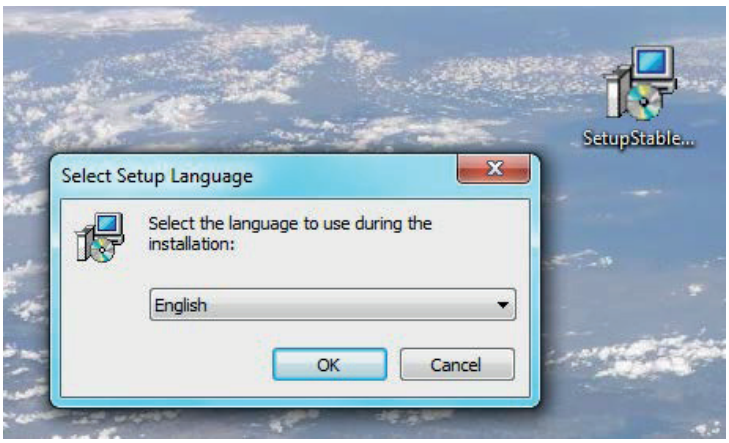
When your .exe pygame program crashes, even though it did not when executed in IDLE
Without IDLE, there is less protection against memory leakage and lost handles. When I experienced problems with crashing Pygame executables it was always due to one thing: the font library. When you have created a font object, you also need to delete it before you go on with the next text. So make sure you always delete variables created with pygame.font after you’ve used them. The font object in Pygame is very sensitive, but apparently Python handles this for you when you run it in the shell, so you will only get into trouble when you have made your executable.

14.2 Making a setup program with e.g. Inno Setup

Distributing your program
To distribute your program you only need to archive your folder e.g to a zip file. You can rename the dist subfolder to mazeman and then create the zip file mazeman.zip, which includes all files and subfolder. In our example about 10 Mb. The user can then unzip this in the Program Files folder and add a shortcut to mazeman.exe on the desktop or on the Start Menu as desired.

There are more fancy ways to distribute your program and create an installer, which will do these same things for you and create an executable for installation. Just download such a program e.g. from download.com. Some are free like NSIS or Inno Setup (last one is recommended).

With Inno Setup, you need to select the 'dist' folder as root and add this (with all files and subfolders). Also you need to indicate which executable is the program to start and perhaps allows an entry in the Start menu. Inno set-up then compiles everything into one neat setup program as a executable e.g. setupmazeman.exe. This is often quite large. Many mail programs or mail servers will prevent sending executables, because that's the way viruses are distributed. So to mail it, you may need to convert this one file to a compressed folder/ archive (ZIP or ARJ extension).



Inno Setup builds professionally looking Setup applications

16. Go exploring: Some pointers for applications of Python beyond this course

15.1 Alternatives to pygame for 2D graphics

15.1.1 Tkinter canvas (included in Python)

Tkinter is primarily meant to design dialog boxes, but also has the so-called canvas object to draw something. See an example below which demonstrates its ease of use:

```
from tkinter import *

master = Tk()
w = Canvas(master, width=200, height=100)
w.pack()
w.create_line(0, 0, 200, 100)
w.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
w.create_rectangle(50, 25, 150, 75, fill="blue")

mainloop()
```

Alternatively, even LOGO like graphics have been included in Python using the turtle module. Not te be used for serious applications but it's really a lot of fun, like a nice, spirograph-like toy and also a very good way to teach younger people how to program!

```
from turtle import *

setup (width=200, height=200, startx=0, starty=0)

speed ("fastest") # important! turtle is intolerably slow otherwise
tracer (False)    # This too: rendering the 'turtle' wastes time

for i in range(200):
    forward(i)
    right(90.5)

done()
```

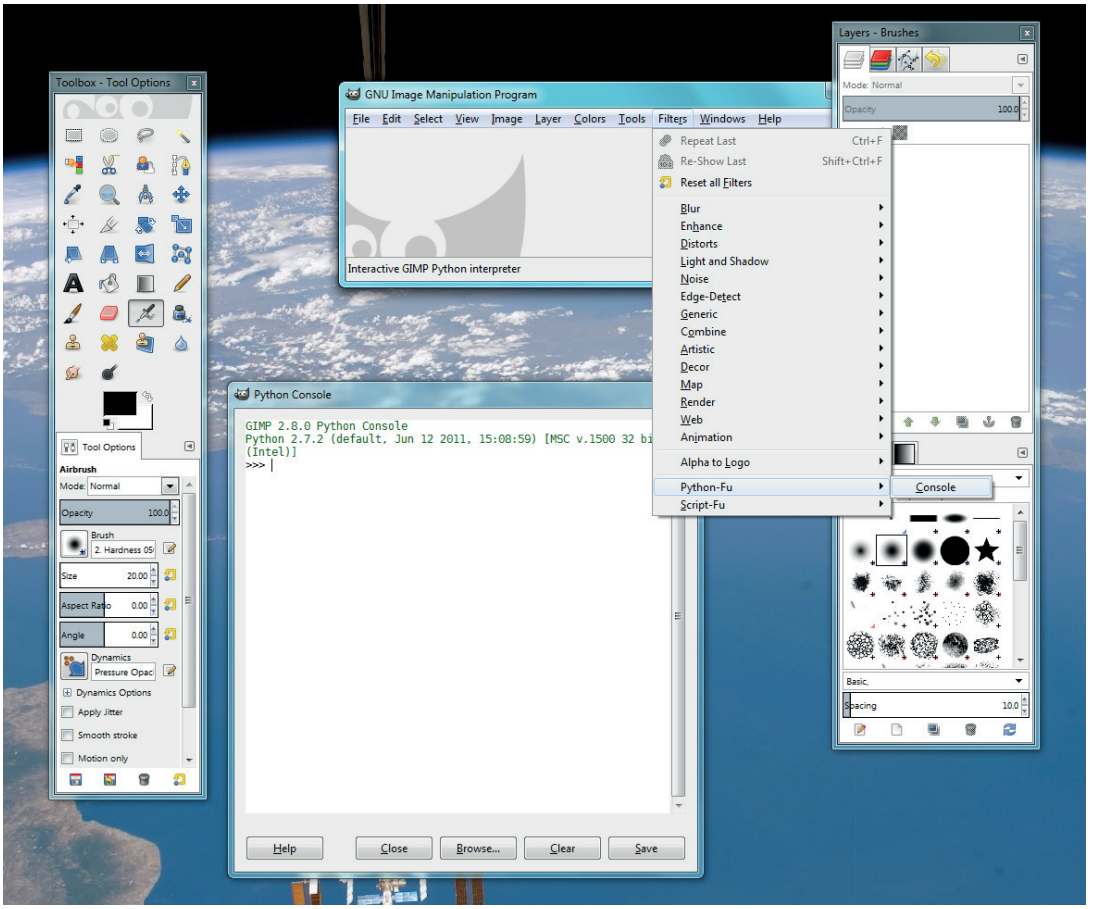
15.1.2 Pycairo (http://cairographics.org/pycairo/)

This is an alternative 2D graphics library for pygame. Differences with pygame are that pycairo seems a bit more advanced but is also slower. The way it works is closer to vector graphics compared to the more pixel oriented pygame library. A nice tutorial can also be found on: http://www.tortall.net/mu/wiki/CairoTutorial

For other options to draw in 2D, see also the windows GUI section, because both PyQT and wxPython have graphics functions included for 2D graphics. This library was discovered and recommended by Fons de Leeuw, a student who used this to visualize acceleration data in his paragliding footage using flight instruments. He found it to produce better images, mainly due to

the anti-aliasing which creates smoother drawing. Because of the vector format, the output quality is also better for e.g. printing.

15.1.3 Using Python Console in GIMP



Not really a graphics library, but the free, open source photoshop program GIMP, has a Python console built-in, which allows you to do batch operations in GIMP. You can find it in the pull down menu “Filters” > “Python FU”. To get an impression of the code an example is given below:

```
g = gimp.pdb
images = gimp.image_list()
my_image = images[0]
layers = my_image.layers

w = g.gimp_image_width(my_image)
h = g.gimp_image_height(my_image)

print("Image Resolution: w=%d,h=%d"%(w,h))

new_layer = g.gimp_layer_new( my_image, w, h, \
          RGBA_IMAGE, "LeopardLayer", 100, NORMAL_MODE)
my_image.add_layer( new_layer )

g.gimp_context_set_pattern("Leopard")
g.gimp_edit_fill(new_layer, PATTERN_FILL)
g.gimp_layer_set_opacity(new_layer, 20)
g.gimp_layer_set_mode(new_layer, SCREEN_MODE)
```

Instruction and a tutorial for Python-FU can be found at : http://www.gimp.org/docs/python/

15.2 Animated 3D graphics

Next to dominating the world of scientific computing and web development, Python is also very large in the professional computer graphics world. Therefore there are many modules to edit images, movies, 3D scenes and sound files.

15.2.1 VPython: easy 3D graphics

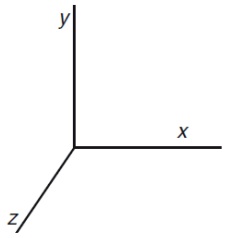
There are a few options to generate moving 3D graphics, depending on to which level you want to take it. Simple, basic 3D shapes can be programmed using VPython (http://www.vpython.org/ where you can also find the documentation). This builds on Pygame and OpenGL (See further below). It however provides a 'layer' over OpenGL to allow even less experienced programmers to build fancy 3D worlds in a relatively simple way. Two starter examples of the code, from the excellent VPython tutorial, are given below, next to their resulting graphical output window. In this window, you can rotate the camera with your mouse when you hold the right mouse button down and you can zoom in/out with both mouse buttons pressed.

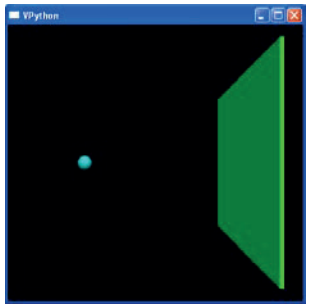
These two lines will result in the window on the right side to be shown:

```
from visual import *

sphere()
```

Objects can be added a simple way (see the axes below to understand the positioning):

```
from visual import *

ball  = sphere(pos=(-5,0,0), radius=0.5, color=color.cyan)

wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
```





15.2.2 Panda3D

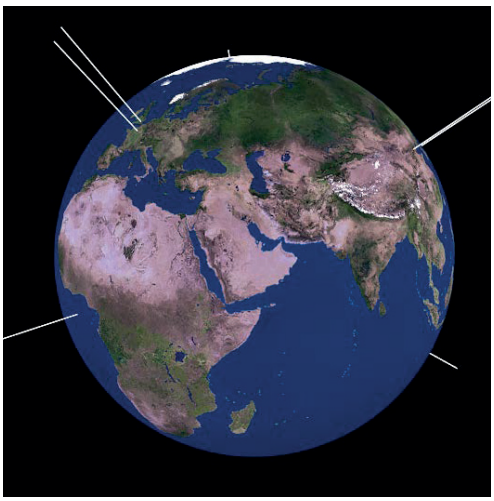
Panda 3D is a very complete 3D graphics and game engine developed orginally and hosted by Carnegie Mellon. This includes very advanced graphics functions. It is compatible with the

PyODE physics engine. Check out their website for some impressive demos: www.panda3d.org and for documentation and download. Panda3D is Open Source and free for any purpose.

15.2.3 Open GL programming

OpenGL has been the standard for 3D graphics for a long time and is at the lowest layer of most other 3D packages. For the Microsoft gamers: it is a sort of cross-platform DirectX. You can then directly conrol the hardware of OpenGl compatible graphics cards. Only to be used by the more daring, demanding and experienced programmer, it provides the possibility to call OpenGL directly from Python with the PyOpenGL module. PyOpenGL is compatible with Pygame: so you can use it in a pygame window (and it often is).

PyOpenGL is very fast and interoperable with a large number of external GUI libraries for Python including wxPython, PyGTK, and Qt. It can also use the GLUT library to provide basic windowing and user interface mechanisms (pulldown menus).



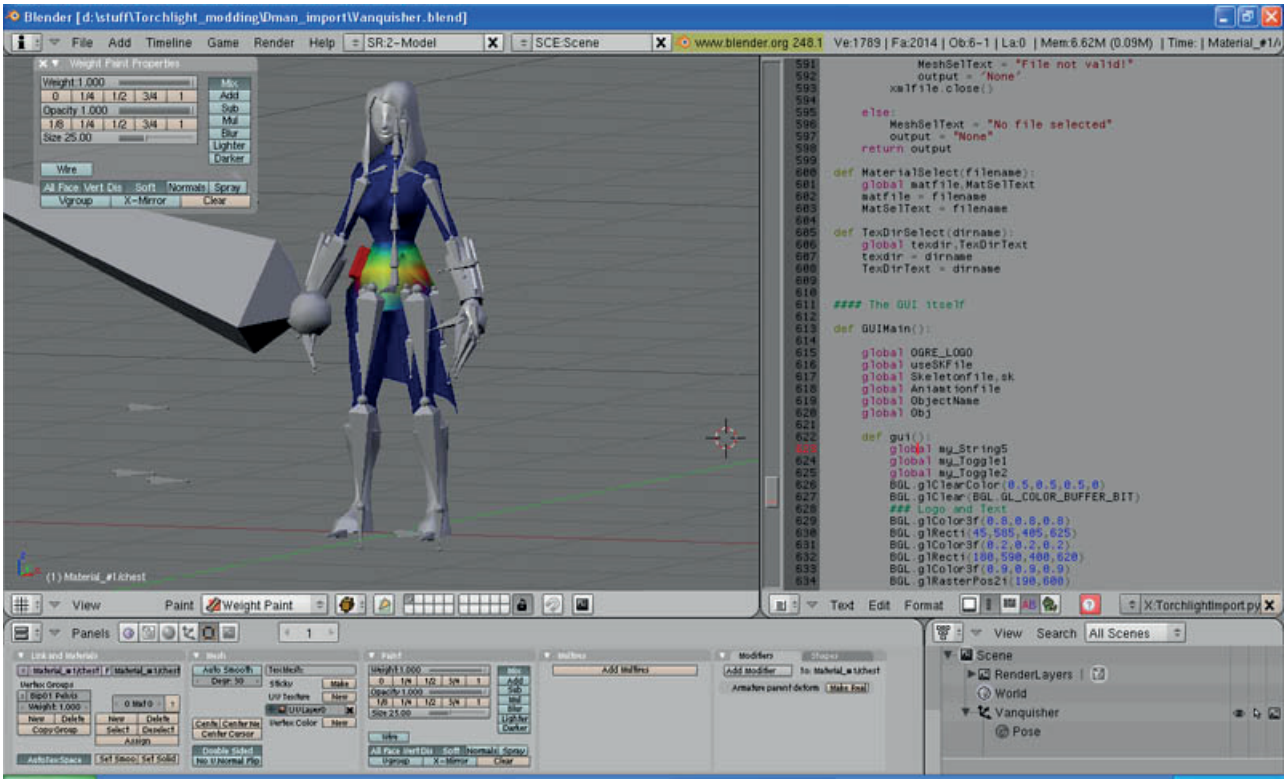
As using OpenGL requires many parameters to be set, example source code tends to be a bit too long to included here as an example. But check out for example:

http://www.willmcgugan.com/blog/tech/2007/6/4/opengl-sample-code-for-pygame/

Or google some examples yourself. The PyOpenGL website has documentation at http://pyopengl.sourceforge.net/documentation/, which will help you to understand the code. You can of course always use some samples as a starting point for your own applications.

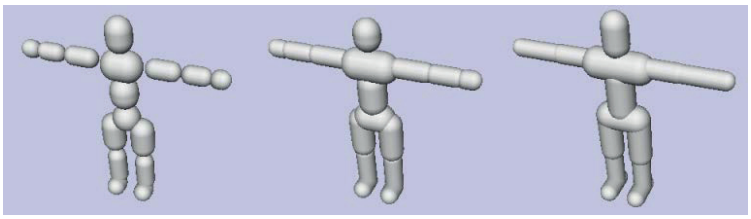
Several students have explored this option and used it successfully to make a 3D solar system visualization, a 3D asteroids game and even a Moonlander simulation on a hilly moon surface. Working with OpenGL is quite complex, so it can be hard, but it is also very powerful.

15.2.3 Blender (www.blender.org)



No matter which 3D library you use, you will need to create 3D objects with complex shapes, surfaces and textures. For this most Python programmers (and many others) use the freeware program Blender. Blender a a very advanced, yet easy to use, 3D content creation suite. You can import objects made in Blender in Python using OpenGL. Even without Python, Blender can make interactive animations. Make sure to visit the website to check out the beautiful gallery. Blender is used a lot by professionals in the Computer graphics world (for commercials, movies and games). Also check out Youtube for some impressive Blender examples. Blender also has a powerful game engine.

15.2.4 Physics Engine: PyODE (http://pyode.sourceforge.net/)



See also the ragdoll demo of PyODE: http://youtu.be/rBolkg1bq4k

To add even more realism, you can also use the PyODE physics engine as your development environment. It provides a simulation of the physics of an extremely rich physical environment. It takes care of realistic mechanics and dynamics of your scenery including gravity, friction, inertia, collisions, etc.. PyODE can be used for simulation, visualisation or gaming. Relatively easy to use, but realize that the closer you get to reality with your simulation, the more complex

your model and your program will become: you'll need to set a lot of parameters, although many defaults are supplied. To get an impression of what it can do check put youtube videos like 'ragdoll demo python ODE'.

15.3 User interfaces: windows dialog boxes, pull-down menus, etc.

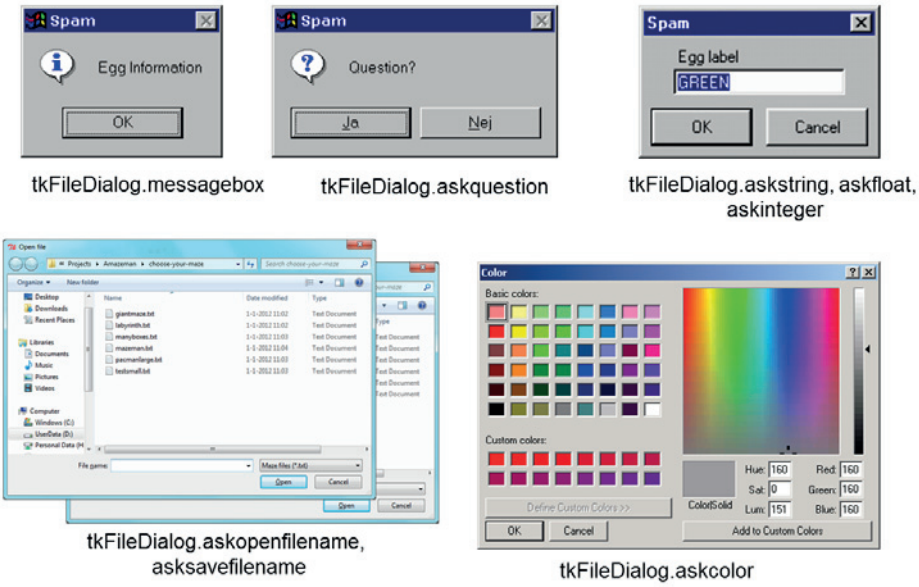
To make your program look professional and easy to use, you might like to add a full-fledged windows interface with pull-down menus and everything. There are several options to do this. Whichever you like or find easy to use really depends on your taste. Some prefer an intuitive editor like provided with PyQt (but with messy code), others like the straightforward typing of Tkinter (with clearer, simple code). Here is the list of options:

15.3.1 Tkinter

Already provided with Python, builds on Tcl/Tk library. The TkInter module is an easy way to use the standard window dialog boxes, e.g. the File Open dialog box (named: `tkFileDialog.askopenfilename`) in the example below:

```
import os,sys
from tkinter import *
import tkinter.filedialog as fd


# Tkinter File Open Dialog for Mazefile
os.chdir('data')      # Move to the subfolder named data
master = Tk()
master.withdraw()     # Hiding tkinter app window

file_path = fd.askopenfilename(title="Open file",
          filetypes=[("Text files",".txt"),("All files",".*")] )

# Quit when user selects Cancel or No file
if file_path == "":
   sys.exit("Ready.")

# Close Tk, return to working directory
master.quit()
os.chdir('..')  # Move back to the main folder
```

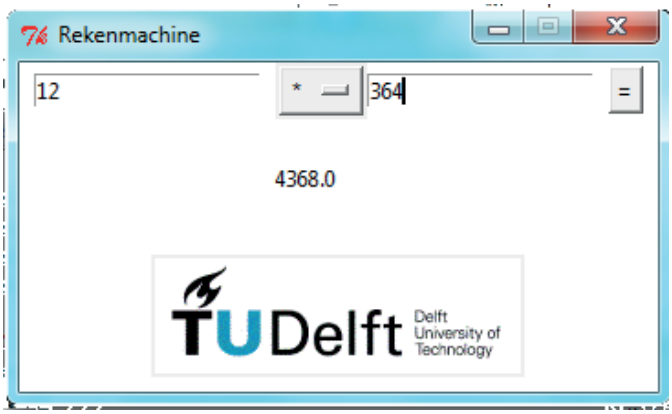
Other standard dialog boxes with their tk name which you can use, like the above example, are:



Even though the Python source will look the same,
the OS determines the actual look of these standard dialogboxes

Some documentation can be found in Python Reference, but more can be found in the pdf file at http://www.pythonware.com/media/data/an-introduction-to-tkinter.pdf. It contains basic functions to build dialog boxes with many controls as well as the handle to call most standard windows dialogs. Easy to use but requires hand-coding and is rather basic in terms of graphics. Still, the IDLE shell and editor you are using, were made in Tkinter. Tcl/Tk has a long-standing record in the Unix world from times far before Python even existed.

An example calculator (see figure below) of which the soure code is given below the figure (made by Eline ter Hofstede):



```
from tkinter import StringVar,Tk,Label,Entry,OptionMenu,Button,PhotoImage

# Change operation according to selection
def changeLabel():
    radioValue = optvar.get()
    if radioValue == '*':
        name =  str(float(str(yourName.get())) * float(str(yourName2.get())))
    if radioValue == '+':
        name =  str(float(str(yourName.get())) + float(str(yourName2.get())))
    if radioValue == '-':
        name =  str(float(str(yourName.get())) - float(str(yourName2.get())))
    if radioValue == '/':
        name =  str(float(str(yourName.get())) / float(str(yourName2.get())))
    labelText.set(name)
    return

# Set up window
app = Tk()
app.title('Rekenmachine')
app.grid()
app.resizable(False,False)

# Create an entry field
custName = StringVar(None)
yourName = Entry(app, textvariable=custName)
yourName.grid(row = 0, column = 0, padx = 8)

# Create an entry field
custName2 = StringVar(None)
yourName2 = Entry(app, textvariable=custName2)
yourName2.grid(row = 0, column = 2)

# Option menu, select the operator
optvar = StringVar()
optvar.set("+")
optionm = OptionMenu(app, optvar, "+","-","/","*").grid(row = 0, column = 1)

# Create an equal to button
button1 = Button(app, text='=',command=changeLabel)
button1.grid(row = 0, column = 3, padx = 8)

# Create a field for the results
labelText = StringVar()
labelText.set('Result')
label1 = Label(app, textvariable=labelText, height=4, bg='white')
label1.grid(row = 1, column = 0,columnspan=3 )

# TUDelft logo
image = PhotoImage(file='tud.gif')
label2 = Label(app, image = image)
label2.grid(column=0,row=2,columnspan=8,pady=8)
app.configure(background='white')

# Run Tkinter main event loop
app.mainloop()
```

15.3.2 PyQt

Qt from Riverbank Computing (: http://www.riverbankcomputing.co.uk/software/pyqt/intro) provides an environment similar to Tkinter. It can also be insatlled with **pip install pyqt** .Comes with many extras, like a QtDesigner, allowing you to graphically draw the dialog boxes. The Spyder editor and IDE were built using PyQt. It builds on Nokia's Qt application framework and runs on all platforms supported by Qt including Windows, MacOS/X and Linux. As a result of using QtDesigner the code is autogenerated and looks less nice. The programming effort then comes down to connecting the right functions to hooks provided by the automatically generated code.

15.3.4 wxPython

Can be found at http://wxpython.org/ Also one of the classics in the Python community, wxPython is fully Open Source, cross-platform (Windows/Linux/Mac OS). I would say it's something in between Tkinter and PyQt in terms of funtionality.

To give an impression of the code, a simple Hello world example is given below. It shows a window called "Hello world" and catches the Close event when the user closes the window to ask for a verification with an OK/Cancel Messagebox.

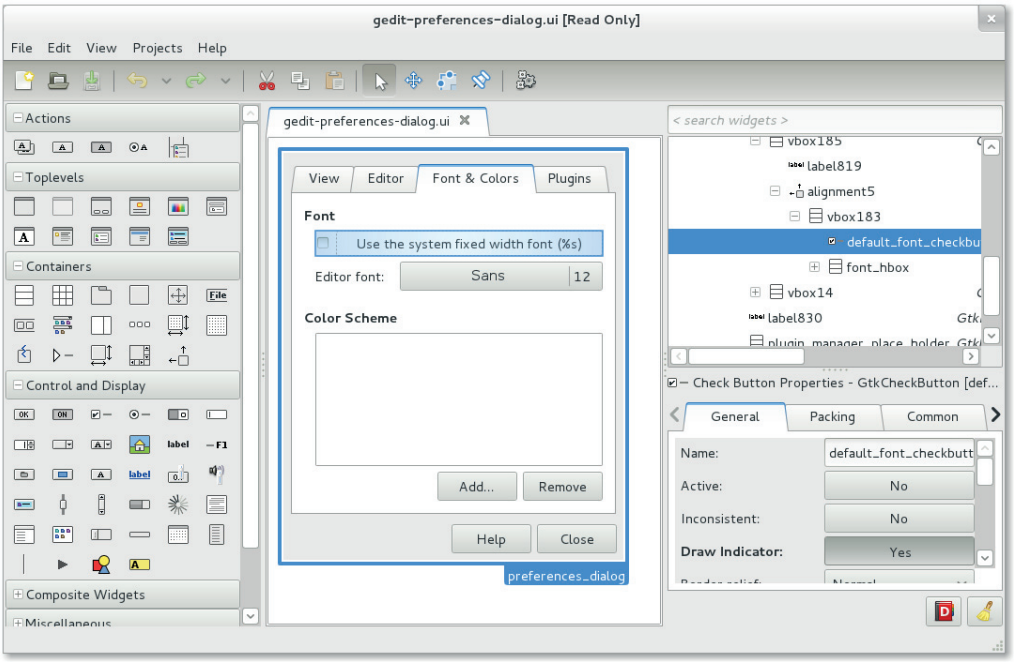
```
import wx

class Frame(wx.Frame):
    def __init__(self, title):
        wx.Frame.__init__(self, None, title=title, size=(350,200))
        self.Bind(wx.EVT_CLOSE, self.OnClose)

    def OnClose(self, event):
        dlg = wx.MessageDialog(self,
            "Are you sure? Do you really want to close this application?",
            "Confirm Exit", wx.OK|wx.CANCEL|wx.ICON_QUESTION)
        result = dlg.ShowModal()
        dlg.Destroy()
        if result == wx.ID_OK:
            self.Destroy()

app = wx.App(redirect=True)
top = Frame("Hello World")
top.Show()
app.MainLoop()
```

15.3.5 GLUT

GLUT, which is a part of OpenGL, is the good old user interface system used by all Open GL fans. Robust, does the job on all platforms, but not always very easy to use.

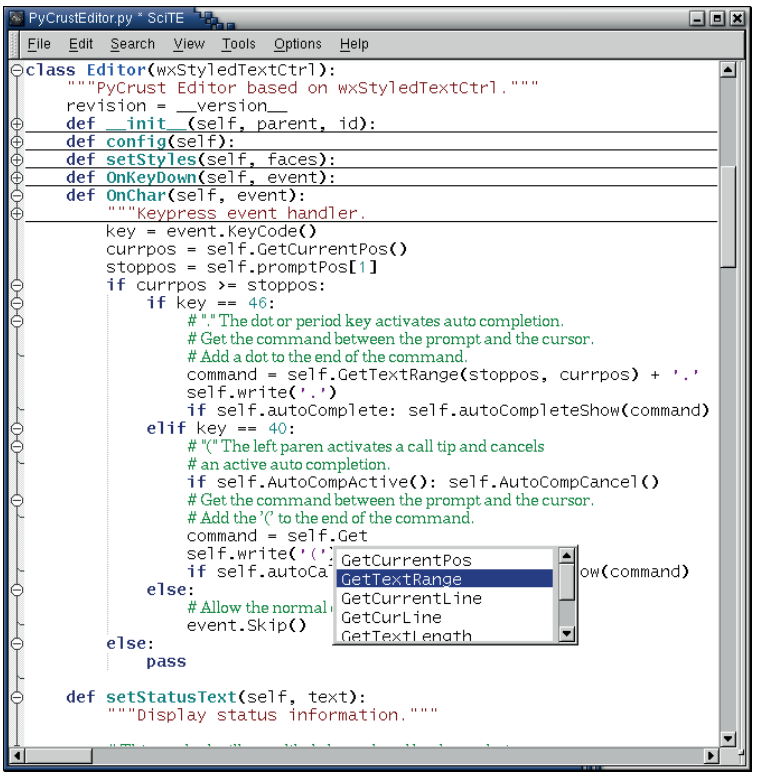
15.3.6 Glade Designer for Gnome

Glade is, like Qt Designer, a tool to graphically edit your dialog boxes and your GUI. It is built on the Gnome desktop and uses the PyGTK library.



PyGTK is not included in python(x,y) but can be downloaded from www.pygtk.org . The Glade program can be found at: http://glade.gnome.org/

As with all of these editors, the result is a mix of generated code (XML data in this case, read by the PyGTK module) and e.g. Python code, which you need to edit yourself. You connect buttons or fields to functions for which you can add the source to add functionality:



The GUI of the Open Source Photoshop program GIMP has been made with the GTK-library.

15.4 Reading from and writing to Excel sheets

As we use MS Office Excel a lot, it is sometimes useful to use their worksheet format from within our Python programs. The easiest way is of course to generate Tab-delimited or comma-delimited data, which can work in both ways. But there are more advanced ways to work directly in Excel spreadsheets using two modules:

Openpyxl: Interfacing with .xlsx and .xlsm files

A very straightforward way to read from and write to excel files is Openpyxl, install with pip install in an Run as Admin console:

```
pip install openpyxl
```

Or use the following link:

http://pythonhosted.org/openpyxl/

An example of the resulting source code if you use this module from the tutorial:

Create your own Workbook:

```
from openpyxl import Workbook

wb = Workbook()
ws = wb.active

# add a simple formula
ws["A1"] = "=SUM(1, 1)"
wb.save("formula.xlsx")
```

Reading from an existing Workbook file:

```
from openpyxl import load_workbook

wb = load_workbook(filename = r'empty_book.xlsx')
sheet_ranges = wb['range names']

print(sheet_ranges['D18'].value) # D18
```

Xlrd and xlwt

A slightly easier way to write and read to Excel is using xlrd and xlwt, install again with pip:

```
pip install xlrd
pip install xlwt
```

One way is to use the xlrd and xlwt module to respectivele read and write to excel sheets. You can find these together with xlutils at http://www.python-excel.org/. An example of the very straightforward calls:

```
from xlwt import *
w = Workbook()
ws = w.add_sheet('F')
ws.write(3, 0, Formula("-(134.8780789e-10+1)"))
w.save('formulas.xls')
```

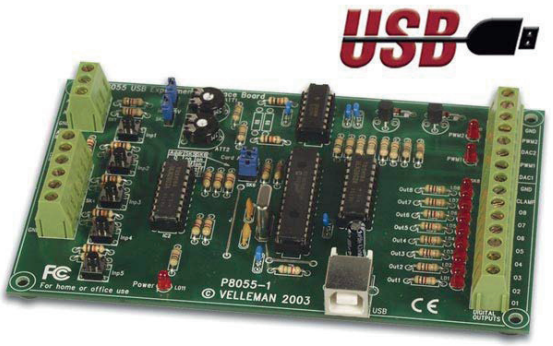
Pyexcelerator
Pyexcelerator is an alternative to xlrd, and xlwt. This module can also be installed with pip:

```
pip install pyexcelerator
```

or it can be found at: http://sourceforge.net/projects/pyexcelerator/ where you can also find docs and examples. The zip file with which you download the module contains many examples for you to borrow.

15.5 Interfacing with hardware

15.5.1 Velleman k8055 example



To let the computer communicate with the external world you might want to connect it to several switches, sensors or analog inputs and outputs. A popular interace board for this, is the Velleman K8055 kit. Student Bastian Telgen has discovered how easy it is to use this with Python and used this at the Lowland festival for Lowlabs.

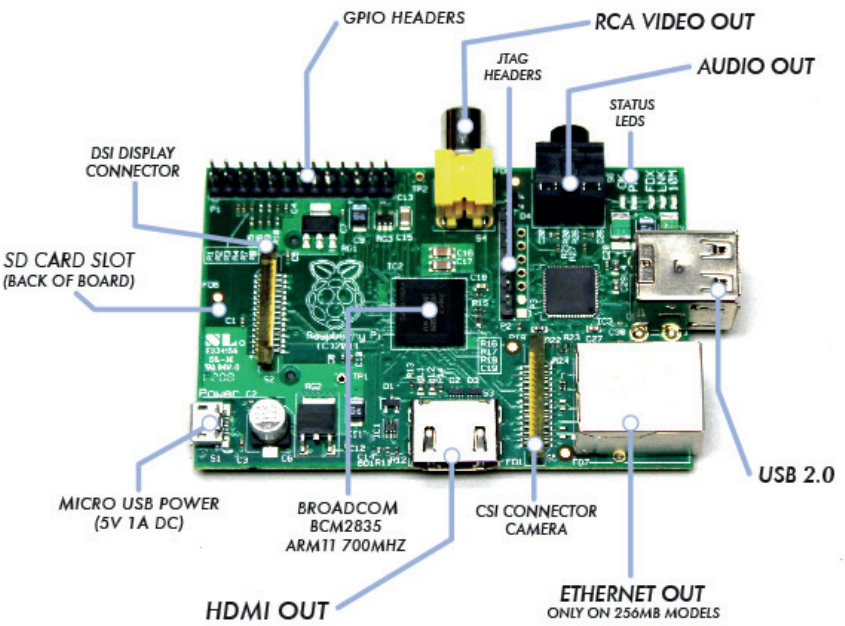
Each Velleman K8055 board has 5 digital (on/off) input channels and 8 digital output channels. There are also two analog input channels and two analog output channels, which all have a 8-bit resolution (so yield a value from 0-255).

A Python interface can be downloaded from the Velleman website: www.velleman.eu. See the example below on how easy it us to use this card from Python. You install the pyk8055 module and connect it to the USB port. You can then run a program like:

```
import pyk8055

dev = pyk8055.device()

dev.digital_on(1)
```

It has been tested on Python 2, I have not yet seen it running on Python 3. So this might mean you might need another PC with Python 2 installed.

15.5.2 Raspberry Pi

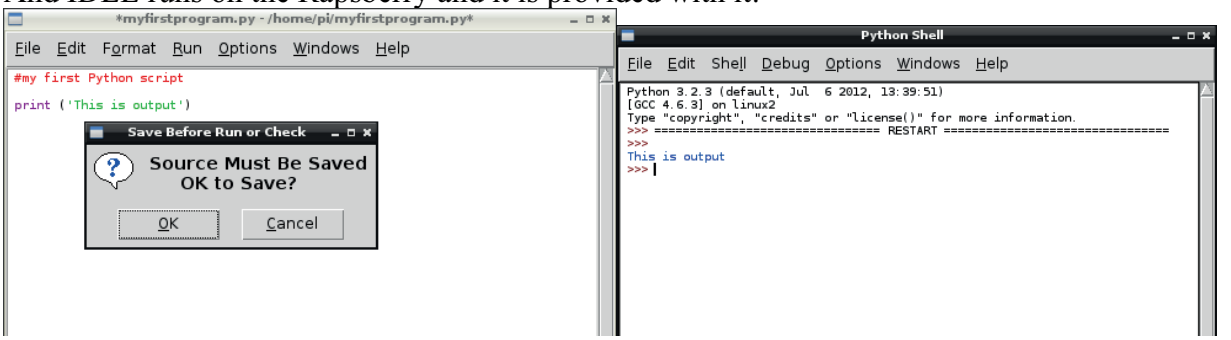


The Raspberry Pi (http://www.raspberrypi.org/) is a very basic (but complete), low cost, very small (credit card size) Linux computer which comes with Python installed and it also runs pygame. For around 50 euros you have a complete computer with the basic accessories like power supply, OS on SD-card, etc. You can connect it to any screen via HDMI. It is used for many purposes like programming/gaming console, mobile webcam, robotics, education projects, twitter and other desktop applications. Some specifications:

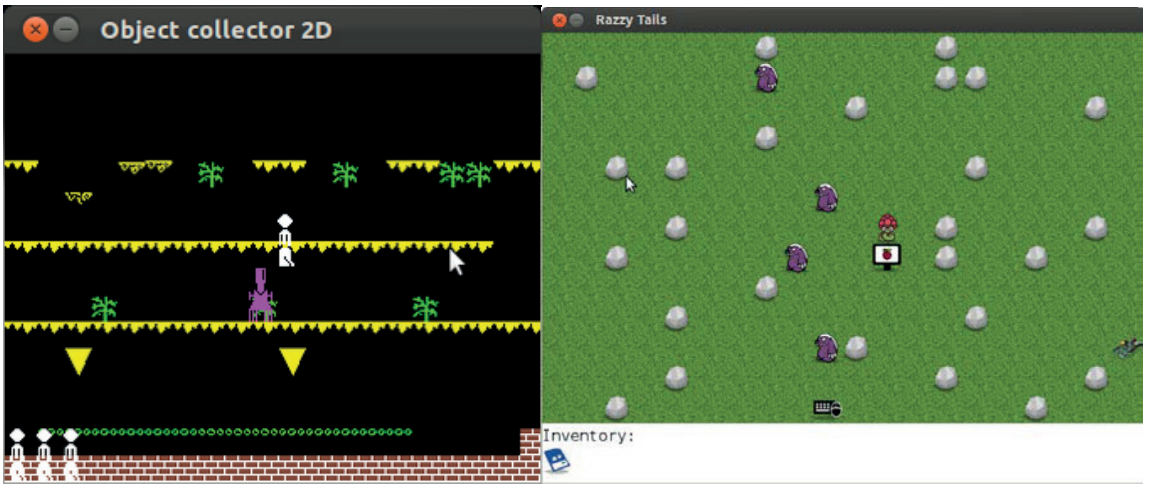
- 700MHz ARM-11 processor
- 256MB of RAM (model 2 has 510 Mb RAM)
- USB 2.0 port
- HDMI Out
- 10/100 Ethernet Port
    - mm audio out

- GPU (1.5 GTexel/s or 24 GFLOPS)
- SD-card (i.s.o. hard disk)

And IDLE runs on the Rapsberry and it is provided with it:

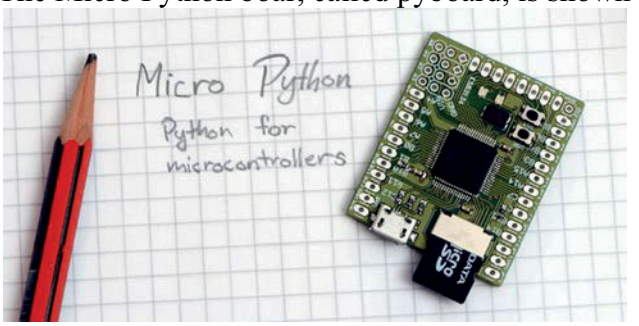


And so does Pygame!



15.5.3 MicroPython

A relatively new initiative (from MIT) is called MicroPython. This is a special version of Python 3 for a microcontroller. The Micro Python boar, called pyboard, is shown below.



By simply copying the text files with your Python code from your PC onto the microboard, you can run the programs. To get an impression of the code, some examples from the website are given below:

Controlling LEDs:

```
led1 = pyb.Led(1)       # get LED 1 object
led2 = pyb.Led(2)       # get LED 2 object
while True:             # loop forever
    mma = pyb.mma()[0]  # get the x-axis angle
    if mma < -10:       # if angled enough one way:
        led1.on()       #     turn LED 1 on
        led2.off()      #     turn LED 2 off
    elif mma > 10:      # if angled the other way:
        led1.off()      #     turn LED 1 off
        led2.on()       #     turn LED 2 on
    else:               # if flat:
        led1.off()      #     turn LED 1 off
        led2.off()      #     turn LED 2 off
    pyb.delay(20)       # wait 20 milliseconds
```

Showing something on LCD screen:

```
# do 1 iteration of Conway's Game of Life
def conway_step():
    for x in range(128):        # loop over x coordinates
        for y in range(32):     # loop over y coordinates
            # count number of neigbours
            num_neighbours = (lcd.get(x - 1, y - 1) +
                lcd.get(x, y - 1) +
                lcd.get(x + 1, y - 1) +
                lcd.get(x - 1, y) +
                lcd.get(x + 1, y) +
                lcd.get(x + 1, y + 1) +
                lcd.get(x, y + 1) +
                lcd.get(x - 1, y + 1))

            # check if the centre cell is alive or not
            self = lcd.get(x, y)

            # apply the rules of life
            if self and not (2 <= num_neighbours <= 3):
                lcd.reset(x, y) # not enough, or too many neighbours: cell dies
            elif not self and num_neighbours == 3:
                lcd.set(x, y)   # exactly 3 neigbours around an empty cell: cell is born


# randomise the start
lcd.clear()                 # clear the LCD
for x in range(128):        # loop over x coordinates
    for y in range(32):     # loop over y coordinates
        if pyb.rng() & 1:   # get a 1-bit random number
            lcd.set(x, y)   # set the pixel randomly

# loop forever, doing iterations of Conway's Game of Life
while True:
    conway_step()           # do 1 iteration
    lcd.show()              # update the LCD
```

More information can be found on the website http://micropython.org/

Appendix A Overview of basic Python statements

`print()`	prints output to console
`input()`	(=function!) returns input from console as text string
`if-elif-else`	conditional execution of statements
`while`	repeat a block of code as long as the condition is True
`for i in range()`	loop n times with i increasing according to specified range
`for x in s`	loop with x equal to each element in list s
`break`	break out of block in current (inner) loop
`continue`	leave to block of code and return to beginning of loop for next iteration
`del a`	delete variable named a (can also be an element of a list: `del tab[2]`)
`def`	define a function
`return` *value*	return from the function to calling program with optional value

`import`	import a module
`from` *module* **`import`** *function*	import specific function from a module
`from` *module* **`import *`**	import all functions from a module in the namespace

`help(`*string***`)`**	get help on module/function in string (`help()` interactive help)
`a = []`	create an empty list named a
`a.append()`	append element to a list named a

File input/output

`f = open(`*filename, mode***`)`**	Open a file, mode can be “r” read, or “w” write.
`f.readline()`	Read a line from a file
`f.writeline(`*line***`)`**	Write a line to a file
`f.close()`	Close the file

Appendix B Overview of functions and special operators

i%k	modulo operator, remainder of i divided by k e.g. 34%10 is 4
xy**	power operator: x^y for integers and floats, so 2**3 is 8 , 2.0**-1.0 is 0.5
txt1 + txt2	concatenates strings txt1 and txt2
s1 + s2	adds list s2 at the end of s1 resulting in combined array
10*[0]	creates a list of 10 zeroes
len(s)	length of a string or list
sum(s)	sum of a list
min(s)	minimum of a list
max(s)	maximum of a list
min(a,b,c)	minimum of given arguments
max(a,b,c)	maximum of given arguments
chr(i)	character (string with length 1) for given ascii code
ord(ch)	ascii code for given character ch
int(x)	converts x to integer
float(i)	converts i to float
str(x)	converts x to a string
eval(txt)	evaluates string resulting in float or integer
s.append(a)	appends a at the end of the list variable s
s.remove(a)	removes the first element with value a from the list variable s
s.index(a)	returns the index of the first appearance of in a list s
range(stop)	produces an iterable list [0,1…..*stop-1*] so until but not incl. *stop*
range(start,stop)	same but now starting with *start* (included) i.s.o. default zero
range(start,stop,step)	same but now with step *step* instead of default 1
math.sqrt(x)	square root of x (needs module math to be imported at beginning of code)
math.exp(x)	returns e^x , also need a math module
random.random()	returns random number (float) between 0.0 and 1.0 (from module random)
random.randint(a,b)	returns random integer with minimum a and maximum b (limits included)
time.clock()	returns clock time as float in seconds (starts at zero the 1st time it is called)
time.time()	returns time tuple with integers: [year, month, date, hour, minute, seconds weekday, yearday, daylightsavingtimeswitch]

Appendix C Example piece of code

```
i = 30              # integer
x = 1.              # float
x = -5.4            # float
x = .3              # float
x = 1e4             # float

sw = True
sw = False              # if lines becomes to long and the line of code
sw = i>40 or x<10       # with a backslash "\" and continue on the next line

s = "Hello"
s = 'World'
s = '"Hello World"'         #Anything after hash-sign is not read
s = "Hello " + "World"      #so this is how you can add comments

print ("Hello world")
print ("Hello",)
print ("World")
print ("Solution is",x," degrees")

a = input("What is your name?")
n = int(input("How many loops?"))
x = float(input("Enter a value for x:"))

a = [1,6,2,7,8,-1]
b = ["Ann","Bernie","Charlie"]

if sw or int(x)>i:
    print("Check done.")
    b = i*2
elif sw:
    print("x is too large")
else:
    print("Something else is going on")

for i in range(10):
    print(i)
    if i > 2:
        print(i*2)

print(range(2,22,2))

i=0
while n!=a[i]:
    i=i+1

ans=""
while not (ans='y' or ans="Y"):
    ans = input("Continue? (Y/N)")

for i in range(10):
    print(i+1)
a = [1,6,2,7,8,-1]
for b in a:
    print(a)
```

Appendix D Solutions to exercises

```
# Exercise  maximum of a,b,c,d
a = float(input('Enter a value a: '))
b = float(input('Enter a value b: '))
c = float(input('Enter a value c: '))
d = float(input('Enter a value d: '))

# Find maximum of a and b
if a>b:
    m1 = a
else:
    m1 = b

# Find maximum of c and d
if c>d:
    m2 = c
else:
    m2 = d

# Print maximum of all
print("Maximum value is:",)

if m1>m2:
    print(m1)
else:
    print(m2)

# Later, you will use one of the built-in functions of Python:
print(max(a,b,c,d))
```

Section 1.4.1

```
#Exercise 1
startsum = float(input("What is your starting amount to save?"))
nyears   = int(input("How many years do you want to save your money?"))
interest = float(input("What is the yearly interest rate (in percent)?"))

endsum = start * (1. + interest/100.)**nyears
print(endsum)


#Exercise 2
Ra = float(input("What is the resistance of resistor A?"))
Rb = float(input("What is the resistance of resistor B?"))
Rc = float(input("What is the resistance of resistor C?"))

Rab = (1./Ra + 1./Rb)**-1

Rabc = Rab + Rc
print(Rabc)
```

Section 1.4.2

```
# Exercise 1
from math import *

a = float(input("Enter the length of right side a: "))
b = float(input("Enter the length of right side b: "))

c = sqrt(a*a + b*b)

print("The length hypothenuse c is",c)
```

Section 1.4.5

```
#Exercise 1:

import random

#0 means heads and 1 means tails
coin = int(random.random()*2)
names = [“heads”,”tails”]

print(names[coin])


#Exercise 2:

import random

#First use random function till 4,simulating the four different playing cards
#Then use a random function till 13, to simulate all the different numbers

icol = int(random.random()*4)
number = int(random.random()*13)

colours = [“hearts”,”tiles”,”clovers”,”pikes”]
print("So your card is", colours[icol], "with ", number)

#Exercise 3:

import time

t = time.localtime()
hour = t.tm_hour
mins = t.tm_min
secs = t.tm_sec

totalsec = hour*3600 + mins*60 + secs

print(“Seconds left in today are:”,24.*3600.-totalsec)
```

Section 2.6.4 Lists

```
#Exercise 1
x = float(input("Enter start amount:"))
interest = float(input("Enter interest rate in percent:"))
nyears = int(input("How many years?"))
rate = 1.0 + interest/100.

debt = []
for i in range(30):
    debt.append([i,round(x)])
    x = x*rate

for i in range(30):
    print(debt[i])
```

Section 3.5 Loops

```
#Exercise 1
value = int(input("To which number do you want to calculate?"))

for i in range(1,int(value)):
    count = 0
    for j in [2]+range(3,i+1,2):
        if i%j == 0:
            count = count+1
    if count <= 2:
        print(i)

#Exercise 2
for i in range(6):
    string = i*"* "
    print(string)

for i in range(4,0, -1):
    string = i*"* "
    print(string)
```

Section 3,6

```
#Exercise 3
words = input("Which string do you want to evaluate?")
d = 0
l = 0

for c in words:
    if c.isdigit():
        d = d + 1
    elif c.isalpha():
        l = l + 1
print("Number of letters is ", l,"Number of digits is ", d)


#Exercise 2
length = int(input("How long do you want your list to be?"))

lst = []
for i in range(length):
    word = input("Enter the word you want in your list?")
    lst.append(word)

for i in range(length):
    print(lst[i], "has length", len(lst[i]))


#Exercise 3
values = [[10, 20], [30, 40, 50, 60, 70]]

for i in range(len(values)):
    print(len(values[i]), values[i])
```

Section 3.7 BREAK or not

```
#Exercise 1

value = int(input("Till which value do you want to calculate the prime
numbers?"))
print ("2 ,",)
for n in range(3, value+1,2):
    for x in range(2, n):
        if n % x == 0:
            break
    else:
        print(n,",",)
print()


#Exercise 1 with while loop and avoid break with a logical used as switch

value = int(input("Till which value do you want to calculate the prime
numbers?"))
print ("2 ,",end=" ")
primes = [2]
for n in range(3,value+1,2):
    swprime = True
    x = 2
    while swprime and x*x<=n:
        if n%x==0:
            swprime = False
        x = x+1+x%2 #add 2 to odd numbers, one to even
    if swprime: print (n,",",end=" ")
print()
```

Chapter 6 exercises

```
Exercises python section 6

#Exercise 1
def countwords(sentence):

    count = 1   #the first word doesn't have a space in front, the rest does
    for i in range(len(sentence)):
        if sentence[i] == " ":
            count = count + 1
            lastspace = i

    lastword = sentence[lastspace+1:]

    return count, lastword

#Exercise 2
def surfacesphere(radius):
    surface = 4.*pi*radius**2
    return surface

def volumesphere(radius):
    volume = (4./3.)*pi*radius**3
    return volume

def surfacecone(radius, height):
    surface = pi*radius*(radius**2 + height**2)**(0.5)
    return surface

def volumecone(radius, height):
    volume = (1./3.)*pi*radius**2*height
    return volume

#Exercise 3
def changestring(string, n, delim):
    string = (n-1)*(string+delim)+string
    return string
```

Section 9.1 Plotting

```
#Exercise 1

import matplotlib.pyplot as plt
import math

xtab = []
fxtab = []
gxtab = []

for i in range(0,314):
    x = float(i)/100.
    fx = math.sin(x)
    gx = math.exp(x) - 2.
    xtab.append(x)
    fxtab.append(fx)
    gxtab.append(gx)

plt.plot(xtab,fxtab)
plt.plot(xtab,gxtab)
plt.show()

#intersection is (1.054, 0.87)

#Exercise 2

import matplotlib.pyplot as plt
import math


xtab = []
ytab = []

for i in range(0,628):
    tetha = float(i)/100.

    r = 4*math.cos(2*tetha)

    x = r*math.cos(tetha)
    y = r*math.sin(tetha)
    xtab.append(x)
    ytab.append(y)

plt.plot(xtab,ytab)
plt.show()

#Exercise 3

import matplotlib.pyplot as plt

time = range(1950,2020,10)
co2 = [250,265,272,280,300,320,389]

plt.plot(time,co2)
plt.show()
```

Section 9.2 Multiple plots

```
#Exercises python section 9.2

import matplotlib.pyplot as plt
import math

#Exercise 1
y1tab = []
y2tab = []
xtab = []
for x in range(0,1000):
    x = x/100.
    xtab.append(x)

    y1 = math.cos(3.*x) + 2.*math.sin(2.*x)
    y1tab.append(y1)

    y2 = math.exp(-2.*x)+ math.exp(-3.*x+1.)
    y2tab.append(y2)

plt.subplot(211)
plt.plot(xtab,y1tab)
plt.subplot(212)
lt.plot(xtab,y2tab)
plt.show()

#Exercise 2

import matplotlib.pyplot as plt
import math

x1tab = []
y1tab = []
x2tab = []
y2tab = []
tethatab = []

for i in range(0,2513):
    tetha = float(i)/100.
    tethatab.append(tetha)

    r1 = tetha
    r2 = tetha**2

    x1 = r1*math.cos(tetha)
    y1 = r1*math.sin(tetha)

    x1tab.append(x1)
    y1tab.append(y1)

    x2 = r2*math.cos(tetha)
    y2 = r2*math.sin(tetha)

    x2tab.append(x2)
    y2tab.append(y2)
```

```
plt.subplot(121)
plt.plot(x1tab,y1tab)
plt.title("r = tetha")
plt.xlabel("x")
plt.ylabel("y")
plt.subplot(122)
plt.plot(x2tab,y2tab)
plt.title("r = tetha**2")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

#Exercise 3

import matplotlib.pyplot as plt

plt.plot([3, 3], [-10, 3], color='k', linestyle='-', linewidth=2)
plt.plot([-10, 3], [3, 3], color='r', linestyle='-', linewidth=2)
plt.plot([-10, 3], [3, -10], color='b', linestyle='--', linewidth=2)

plt.show()
```

Section 9.3 Inter-active plots

```
#Exercises python section 9.3

import matplotlib.pyplot as plt

plt.ion()

for x in range(-20,20):
    x = float(x)/10.
    y1 = (2**2 - x**2)**0.5
    y2 = -(2**2 - x**2)**0.5

    plt.plot(x,y1, 'ro')
    plt.plot(x,y2, 'ro')
    plt.draw()

plt.close()
```

Section 9.4 3D-plots

```
from matplotlib import cm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy import arange, meshgrid, linspace

#Exercise 1
x = arange(0,21,1)
y = linspace(0,8,21)
X, Y = meshgrid(x, y)
Z = (1./100.)*X**2

fig = plt.figure()
ax = Axes3D(fig)

ax.plot_surface(X,Y,Z, rstride=1, cstride=1, linewidth=0, antialiased=False)
plt.show()
```

Chapter 10 exercises: Simulation

```
import matplotlib.pyplot as plt
import math

#Exercise 1 a), b), c)
t = 0.                #[s]     time starts at zero
dt= 0.1               #[s]     time step
vy = 0.               #[m/s]   starting speed is zero
y = 10.               #[m]     start altitude
g = 9.81              #[m/s2] gravity acceleration
m = 2.0               #[kg]    mass
ttab = []
ytab = []

CD = 0.47
rho = 1.225
S = math.pi*0.15**2

while y > 0:
    t = t + dt
    F = m * g
    D = 0.5*CD*rho*vy**2*S
    a = (-1.*F + D) / m
    vy = vy + a*dt
    y = y + vy*dt

    ttab.append(t)
    ytab.append(y)

plt.plot(ttab,ytab)
plt.show()
```

```
#Exercise 2 a)
delta = 0.01
s = 0
h = 1.
tetha = 30.*math.pi/180.
vx = 100*math.cos(tetha)
vy = 100*math.sin(tetha)
g = 9.81
stab = []
htab = []
while h >= 0:
    ay = -1.*g
    ax = 0

    vy = vy + ay*delta
    vx = vx + ax*delta

    h = h + vy*delta
    s = s + vx*delta

    htab.append(h)
    stab.append(s)

plt.plot(stab,htab)
plt.ylim(0,140)
plt.show()

#Exercise 2 b)
delta = 0.01
s = 0
h = 1.
tetha = 30.*math.pi/180.
vx = 100*math.cos(tetha)
vy = 100*math.sin(tetha)
g = 9.81
Fd = 80
m = 10
stab = []
htab = []
while h >= 0:
    ay = (-1.*m*g -1.*Fd*math.sin(tetha))/m
    ax = (-1*Fd*math.cos(tetha))/m

    vy = vy + ay*delta
    vx = vx + ax*delta

    h = h + vy*delta
    s = s + vx*delta

    htab.append(h)
    stab.append(s)

    tetha = math.atan(vy/vx)

plt.plot(stab,htab)
plt.ylim(0,140)
plt.show()
```

```
#Exercise 2 c)
delta = 0.01
s = 0
h = 1.
tetha = 30.*math.pi/180.
vx = 100*math.cos(tetha)
vy = 100*math.sin(tetha)
m = 10
g0 = 9.80665
Re = 6371*1000 #m
stab = []
htab = []
while h >= 0:
    Fd = 0.05*(vx**2+vy**2)
    g = g0*(Re/(Re + h))

    ay = (-1.*m*g -1.*Fd*math.sin(tetha))/m
    ax = (-1*Fd*math.cos(tetha))/m

    vy = vy + ay*delta
    vx = vx + ax*delta

    h = h + vy*delta
    s = s + vx*delta

    htab.append(h)
    stab.append(s)

plt.plot(stab,htab)
plt.ylim(0,140)
plt.show()
```

Section 11.2 Exercises

```
#Exercises python section 11.2

from numpy import *
import matplotlib.pyplot as plt

#Exercise 1
#Make functions for the circle:
r = 2

tetha = linspace(0,pi,180)
x1 = r*cos(tetha)
y1 = r*sin(tetha)
x2 = -r*cos(tetha)
y2 = -r*sin(tetha)

#Make a list with all the points of the hexagon
xlist = [-r, -r*cos(pi/3.), r*cos(pi/3.), r, r*cos(pi/3.), -r*cos(pi/3.), -r]
ylist = [0, r*sin(pi/3.), r*sin(pi/3.), 0, -r*sin(pi/3), -r*sin(pi/3), 0]

plt.plot(x1,y1, 'r-', x2,y2, 'r-')
plt.plot(xlist,ylist, 'b-')
plt.show()

#Exercise 2
def diagonal(array):
    maintab = []
    for i in range(len(array)):
        main = array[i,i]
        maintab.append(main)

    return maintab

A = ones((2,2))
A[0,1] = A[0,1]*5
A[0,0] = A[0,0]*2
A[1,1] = A[1,1]*-1
A[1,0] = A[1,0]*4
print A


#Exercise 3
#list
lst = [[2,3],[5,4]]

column = []
for i in range(len(lst)):
    col = lst[i][0]
    column.append(col)
print(column)

arr = array(lst)

print(arr[:,0])
```

Section 11.3 exercises Logic and arrays

```
import numpy as np
import matplotlib.pyplot as plt

#Exercise 1
h    = np.arange(0,32.,1.)

to11 = 288.15 - 6.5*h
to20 = 216.65+0.*(h-11.)
to32 = 216.65+1.*(h-20.)

temp = (h<11)*to11 + (h>=11)*to20 + (h>=20)*to32 -(h>=20)*to20

plt.plot(h, temp)
plt.show()

#Exercise 2
x = np.arange(-3,3,0.001)

fx = (x<=1.)*x**2 + (x>1)*(6.-x)

plt.plot(x,fx)
plt.show()

#Exercise 3
A = np.array([3,4,6,10,24,89,45,43,46,99,100])

div3 = A[A%3!=0]
print("Elements of A not divisible by 3:", div3)

div5 = A[A%5==0]
print("Elements of A divisible by 5:", div5)

div35 = A[(A%3==0) & (A%5==0)]
print("Elements of A, which are divisible by 3 and 5:", div35)

A[A%3==0] = 42
print ("New values of A after setting the elements of A,",      \
       "which are divisible by 3, to 42:", A)
```

Section 11.5

```
from numpy import *
import matplotlib.pyplot as plt

#Exercise 1
A = matrix('2 -5 6; 4 3 8; 5 -2 9')
b = matrix('3; 5; -1')

B = A.I

x = B*b

print(x)

#Exercise 2
#Solution: u is motorboat speed and v is current speed

# (3 + 10/60) *(u-v) = 29.5 ->          u - v = (29.5/3.16667)
# (2 + 6/60)*(u+v) = 29.5 ->            u + v = (29.5/2.1)

A = matrix('1 -1; 1, 1')
b = matrix('9.31578; 14.0476')
B = A.I

x = B*b
print(x)

#Exercise 3
#First we calculate all the data points
i = arange(1,11)
xi = 0.1*i
yi = 5*exp(-xi)+2*xi
Yi = ones(shape=(10,1))
Yi[:,0] = Yi[:,0]*yi

#Now we use least square problem to make a fitting. We want a lineair trend
line, so
#y = b1 + xb2 so Xb = y with

#First construct matrix X:

X = ones(shape=(10,2))
X[:,1] = X[:,1]*xi
X = matrix(X)

#Now we want to generate a least square solution for fitting! X.TXb = X.Ty

Xtranspose = X.T
Xfirst = Xtranspose*X
Xsecond = Xtranspose*Yi

#Solve for b to get the fit
b = linalg.inv(Xfirst)*Xsecond

#Or calculate it faster with the function in numpy.linalg:
c = linalg.lstsq(X,yi) #Check if this gives the same result!
```

```
#So the least square fit is:
y = float(b[0][0]) + float(b[1][0])*xi

#Now plot both curves
plt.plot(xi,yi,label='Data points')
plt.plot(xi,y,label='Least square fit')
plt.legend()
plt.show()
```

Exercises chapter 13

```
#Exercise 1

import pygame as pg

pg.init()

reso = (750,500)
blue = (0,0,255)

screen = pg.display.set_mode(reso)
screen.fill(blue)

pg.display.flip()

pg.event.pump()

pg.quit()


#Exercise 2
import pygame as pg

pg.init()

reso = (750,500)
blue = (0,0,255)

screen = pg.display.set_mode(reso)
screen.fill(blue)

plane = pg.image.load('plane.jpg')
plane = pg.transform.scale(plane,(150,60))
planerect = plane.get_rect()

planerect.center = (75,250.)
screen.blit(plane,planerect)

pg.display.flip()

pg.event.pump()

pg.quit()
```

```
#Exercise 3

import pygame as pg

pg.init()

reso = (750,500)
blue = (0,0,255)

screen = pg.display.set_mode(reso)
scrrect = screen.get_rect()

plane = pg.image.load('plane.jpg')
plane = pg.transform.scale(plane,(150,60))
planerect = plane.get_rect()

xs = 75
ys = 250
vx = 50
tsim  = 0.0
tstart =  0.001*pg.time.get_ticks()
dt = 0.01
running = True

while running:
    trun = 0.001*pg.time.get_ticks() - tstart
    if trun + dt >= tsim:
        tsim = tsim + dt
        xs = xs + vx*dt
        planerect.center = (xs,ys)

    pg.draw.rect(screen,blue,scrrect)
    screen.blit(plane,planerect)

    pg.display.flip()

    if xs > 750+75:
        running = False

    pg.event.pump()

pg.quit()
print("Ready.")
```

```
#Exercise 4
import pygame as pg

pg.init()

reso = (750,500)
blue = (0,0,255)

screen = pg.display.set_mode(reso)
scrrect = screen.get_rect()

plane = pg.image.load('plane.jpg') # use jpg of a plane flying to the right
plane = pg.transform.scale(plane,(150,60))
planerect = plane.get_rect()

xs = 75
ys = 250
vx = 50
tsim  = 0.0
tstart =  0.001*pg.time.get_ticks()
dt = 0.01
running = True

while running:
    trun = 0.001*pg.time.get_ticks() - tstart
    if trun + dt >= tsim:
        tsim = tsim + dt
        xs = xs + vx*dt

    if xs > 750+75:
        running = False

    for event in pg.event.get():
        if event.type == pg.QUIT:
            running = False
        if event.type == pg.KEYDOWN:
            if event.key == pg.K_ESCAPE:
                running = False
            elif event.key == pg.K_DOWN:
                ys = ys + 5
            elif event.key == pg.K_UP:
                ys = ys - 5

    planerect.center = (xs,ys)
    pg.draw.rect(screen,blue,scrrect)
    screen.blit(plane,planerect)

    pg.display.flip()


pg.quit()
```