Basic Python by examples

1. Python installation

On Linux systems, Python 2.x is already installed.

To download Python for Windows and OSx, and for documentation see http://python.org/

It might be a good idea to install the very useful modules Numpy, Scipy and Matplotlib which will be used later in tutorial.

2. Python 2.x or Python 3.x ?

The current version is 3.x

Some libraries may not yet be available for version 3, and Linux Ubuntu comes with 2.x as a standard. Many approvements from 3 have been back ported to 2.7.

The main differences for basic programming are in the print and input functions.

We will use Python 2.x in this tutorial.

3. Python interactive: using Python as a calculator

Start Python (or IDLE, the Python IDE).
A prompt is showing up:

```
>>>
```

Display version:

```
>>>help()
Welcome to Python 2.7!  This is the online help utility.
...
help>
```

Help commands:

modules:	available modules
keywords:	list of reserved Python keywords
quit:	leave help

To get help on a keyword, just enter it's name in help.

Simple calculations in Python

```
>>> 3.14*5
15.700000000000001
```

Supported operators:

Operator		Example	Explication
+, - *, /	add, substract, multiply, divide		
%	modulo	25 % 5 = 0 84 % 5 = 4	25/5 = 5, remainder = 0 84/5 = 16, remainder = 4
**	exponent	2**10 = 1024	
//	floor division	84//5 = 16	84/5 = 16, remainder = 4

Take care in Python 2.x if you divide two numbers:
Isn't this strange:

```
>>> 35/6
5
```

Obviously the result is wrong!
But:

```
>>> 35.0/6
5.833333333333333
>>> 35/6.0
5.833333333333333
```

In the first example, 35 and 6 are interpreted as integer numbers, so integer division is used and the result is an integer.
This uncanny behavior has been abolished in Python 3, where 35/6 gives 5.833333333333333.

In Python 2.x, use floating point numbers (like 3.14, 3.0 etc....) to force floating point division!

Another workaround would be to import the Python 3 like division at the beginning:

```
>>> from __future__ import division
>>> 3/4
0.75
```

Builtin functions:

```
>>> hex(1024)
'0x400'

>>> bin(1024)
'0b10000000000'
```

Expressions:

```
>>> (20.0+4)/6
4
>>> (2+3)*5
25
```

4. Using variables

To simplify calculations, values can be stored in variables, and and these can be used as in normal mathematics.

```
>>> a=2.0
>>> b = 3.36
>>> a+b
5.359999999999999
>>> a-b
-1.3599999999999999
>>> a**2 + b**2
15.289599999999998
>>> a>b
False
```

The name of a variable must not be a Python keyword!

Keywords are:

```
and         elif        if          print
as          else        import      raise
assert      except      in          return
break       exec        is          try
class       finally     lambda      while
continue    for         not         with
def         from        or          yield
del         global      pass
```

5. Mathematical functions

Mathematical functions like square root, sine, cosine and constants like pi etc. are available in Python. To use them it is necessary to import them from the math module:

```
>>> from math import *
>>> sqrt(2)
1.4142135623730951
```

Note:
There is more than one way to import functions from modules. Our simple method imports all functions available in the math module. For more details see appendix.

Other examples using math:

Calculate the perimeter of a circle

```
>>> from math import *
>>> diameter = 5
>>> perimeter = 2 * pi * diameter
>>> perimeter
31.41592653589793
```

Calculate the amplitude of a sine wave:

```
>>> from math import *
>>> Ueff = 230
>>> amplitude = Ueff * sqrt(2)
>>> amplitude
325.2691193458119
```

6. Python scripts (programs)

If you have to do more than a small calculation, it is better to write a script (a program in Python).

This can be done in IDLE, the Python editor.
A good choice is also Geany, a small freeware editor with syntax colouring, from which you can directly start your script.

To write and run a program in IDLE:

- Menu File – New Window
- Write script
- File – Save (name with extension .py, for example myprogram.py)
- Run program: <F5> or Menu Run – Run Module

Take care:

- In Python white spaces are important!
  The indentation of a source code is important!
  A program that is not correctly indented shows either errors or does not what you want!

- Python is case sensitive!
  For example x and X are two different variables.

7. A simple program

This small program calculates the area of a circle:

```
from math import *
d = 10.0                                    # diameter
A = pi * d**2 / 4 print
"diameter =", d print "area =
", A
```

Note: everything behind a "#" is a comment.
Comments are important for others to understand what the program does (and for yourself if you look at your program a long time after you wrote it).

8. User input

In the above program the diameter is hard coded in the program.
If the program is started from IDLE or an editor like Geany, this is not really a problem, as it is easy to edit the value if necessary.

In a bigger program this method is not very practical.

This little program in Python 2.7 asks the user for his name and greets him:

```
s = raw_input("What is your name?")
print "HELLO ", s
```

```
What is your name?Tom
HELLO  Tom
```

Take care:
The raw_input function gives back a string, that means a list of characters. If the input will be used as a number, it must be converted.

9. Variables and objects

In Python, values are stored in objects.
If we do

```
d = 10.0
```

a new object d is created. As we have given it a floating point value (10.0) the object is of type floating point. If we had defined d = 10, d would have been an integer object.

In other programming languages, values are stored in variables. This is not exactly the same as an object, as an object has "methods", that means functions that belong to the object.
For our beginning examples the difference is not important.

There are many object types in Python.

The most important to begin with are:

Object type	Type class name	Description	Example
Integer	int	Signed integer, 32 bit	a = 5
Float	float	Double precision floating point number, 64 bit	b = 3.14
Complex	complex	Complex number	c = 3 + 5j c= complex(3,5)
Character	chr	Single byte character	d = chr(65) d = 'A' d = "A"
String	str	List of characters, text string	e = 'LTAM' e = "LTAM"

10. Input with data conversion

If we use the raw_input function in Python 2.x or the input function in Python 3, the result is always a string. So if we want to input a number, we have to convert from string to number.

```
x = int(raw_input("Input an integer: ")) y =
float(raw_input("Input a float: ")) print x, y
```

Now we can modify our program to calculate the area of a circle, so we can input the diameter:

```
""" Calculate area of a circle"""
from math import *
d = float(raw_input("Diameter: ")) A = pi *
d**2 / 4
print "Area = ", A
```

```
Diameter: 25
Area = 490.873852123
```

Note:
The text at the beginning of the program is a description of what it does. It is a special comment enclosed in triple quote marks that can spread over several lines.
Every program should have a short description of what it does.

11. While loops

We can use the computer to do tedious tasks, like calculating the square roots of all integers between 0 and 100. In this case we use a while loop:

```
""" Calculate quare root of numbers 0 to 100"""
from math import *

i = 0
while i<= 100:
      print i, "\t\t" ,    sqrt(i)
      i = i + 1
print "READY!"
```

```
0             0.0
1             1.0
2             1.41421356237
3             1.73205080757
.....
98            9.89949493661
99            9.94987437107
100           10.0
READY!
```

The syntax is :

```
while <condition> :
        <....
        block of statements
        ...>
```

The block of statements is executed as long as <condition> is True, in our example as long as i <= 100.

Take care:

- Don't forget the ":" at the end of the while statement
- **Don't forget to indent the block that should be executed inside the while loop!**

The indentation can be any number of spaces (4 are standard), but it must be consistent for the whole block.

Avoid endless loops!

In the following example the loop runs infinitely, as the condition is always true:

```
i = 0
while i<= 5 :
       print i
```

The only way to stop it is by pressing <Ctrl>-C.

Examples of conditions:

Example	
`x == 3`	True if x = 3
`x != 5`	True if x is not equal to 5
`x < 5` `x > 5`	
`x <= 5` `x >= 5`	

Note:
i = i +1 can be written in a shorter and more "Pythonic" way as i += 1

12. Testing conditions: if, elif, else

Sometimes it is necessary to test a condition and to do different things, depending on the condition.
Examples: avoiding division by zero, branching in a menu structure etc.

The following program greets the user with "Hello Tom", if the name he inputs is Tom:

```
s = raw_input ("Input your name: ")

if s == "Tom":
    print "HELLO ", s
```

Note the indentation and the ":" behind the if statement!

The above program can be extended to do something if the testing condition is not true:

```
s = raw_input ("Input your name: ")
if s == "Tom":
    print "Hello ", s else:
    print "Hello unknown"
```

It is possible to test more than one condition using the elif statement:

```
s = raw_input ("Input your name: ")
if s == "Tom":
    print "Hello ", s elif s
== "Carmen":
    print "I'm so glad to see you ", s elif s ==
"Sonia":
    print "I didn't expect you ",s else:
    print "Hello unknown"
```

Note the indentation and the ":" behind the if, elif and else statements!

13. Tuples

In Python, variables can be grouped together under one name. There are different ways to do this, and one is to use tuples.

Tuples make sense for small collections of data, e.g. for coordinates:

```
(x,y) = (5, 3) coordinates
= (x,y) print coordinates

dimensions = (8, 5.0, 3.14)
print dimensions print
dimensions[0] print
dimensions[1] print
dimensions[2]
```

```
(5, 3)
(8, 5.0, 3.14)
8
5.0
3.14
```

Note:
The brackets may be omitted, so it doesn't matter if you write x, y or (x, y)

14. Lists (arrays)

Lists are ordered sequences of objects.
It can for example be very practical to put many measured values, or names of an address book, into a list, so they can be accessed by one common name.

```
nameslist = ["Sam", "Lisy", "Pit"]
numberslist = [1, 2, 3.14]
mixedlist = ["ham", 'eggs', 3.14, 5]
```

Note:
Unlike other programming languages Python's arrays may contain different types of objects in one list.

New elements can be appended to a list:

```
a=[0,1,2]
print a

a.append(5) a.append(
"Zapzoo") print a
```

```
[0, 1, 2]
[0, 1, 2, 5, 'Zapzoo']
```

An empty list can be created this way:

```
x=[]
```

Sometimes we need an array that is initialized with zero values.

This is done with:

```
y= [0]*10          # array of integers with 10 zero elements
z = [0.0]*20       # array of floats with 20 zero elements
```

The number of elements can be determined with the len (length) function:

```
a=[0,1,2]
print  len(a)
```

```
3
```

The elements of a list can be accessed one by one using an index:

```
mylist = ["black", "red", "orange"]
print    mylist[0] print
mylist[1]          print
mylist[2]
```

```
black
red
orange
```

15. Range: producing lists of integer numbers

Often you need a regularly spread list of numbers from a beginning value to an end value.

This is done by the range function:

```
""""  range gives a list of int numbers
        note that end value is NOT included!  """

r1 =  range(11)                 # 0...10
print  r1                         # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

r2 =  range(5,16)               # 5...15
print  r2                         # [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

r3 =  range(4,21,2)           # 4...20    step 2
print  r3                         # [4, 6, 8, 10, 12, 14, 16, 18, 20]

r4 =  range(15, 4, -5)      # 15....5 step -5
print  r4                         # [15, 10, 5]
```

The general syntax is

```
range (<startvalue>, <endvalue>, <stepsize>)
```

Take care:

- A strange (and somewhat illogical) detail of the range function is that the end value is excluded from the resulting list!
- The range function only works for integers!

16. Producing lists of floating point numbers

If you need floating point numbers, use linspace from the Numpy module, a package that is very useful for technical and scientific applications. This package must be installed first, it is available at http://www.numpy.org/
Don't forget to import the module in your script!

Note:
Here we use a slightly different method of import that avoids confusion between names of variables and numpy funtions. There are 3 ways to import functions from a module, see appendix.

```
""" for floating point numbers use linspace and logspace from numpy!"""
import numpy as np
r5 = np.linspace(0,2,9)
print r5
```

```
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75  2.  ]
```

The syntax for linspace is

```
linspace ( <startvalue>, <stopvalue>, <number_of_values> )
```

The next example gives 9 logarithmically spaced values between 100 = 10^2 and 1000 = 10^3:

```
r6 = np.logspace(2, 3, 9)
print r6
```

```
[  100.           133.35214322   177.827941     237.13737057   316.22776602
   421.69650343   562.34132519   749.89420933  1000.        ]
```

17. Iterating through a list: the for loop

If we have to do something with all the elements of a list (or another sequence like a tuple etc.) one after the other, we use a for loop.

The example uses the names of a list of names, one after one:

```
mynames = [ "Sam", "Pit", "Misch" ]

for n in mynames:
    print "HELLO ", n
```

```
HELLO  Sam
HELLO  Pit
HELLO  Misch
```

This can also be done with numbers:

```
from math import *
for i in range (0, 5):
        print i, "\t", sqrt(i)
```

```
0       0.0
1       1.0
2       1.41421356237
3       1.73205080757
4       2.0
```

Notes:

- Python's for loop is somewhat different of the for ... next loops of other programming languages. in principle it can iterate through anything that can be cut into slices. So it can be used on lists of numbers, lists of text, mixed lists, strings, tuples etc.

- In Python the for ... next construction is often not to be missed, if we think in a "Pythonic" way. Example: if we need to calculate a lot of values, it is not a good idea to use a for loop, as this is very time consuming. It is better to use the Numpy module that provides array functions that can calculate a lot of values in one bunch (see below).

18. Iterating with indexing

Sometimes you want to iterate through a list and have access to the index (the numbering) of the items of the list.

The following example uses a list of colour codes for electronic parts and prints their index and the colour. As the colours list is well ordered, the index is also the colour value.

```
""" Dispay resistor colour code values"""
colours = [ "black", "brown", "red", "orange", "yellow", "green", "blue",
                "violet", "grey","white" ]

cv = list (enumerate (colours))

for c in cv:
    print c[0], "\t", c[1]
```

The list(enumerate (....)) function gives back a list of tuples cv that contain each an index (the numbering) and the colour value as text. If we print this we see

```
[(0, 'black'), (1, 'brown'), (2, 'red'), (3, 'orange'), (4, 'yellow'), (5,
'green'), (6, 'blue'), (7, 'violet'), (8, 'grey'), (9, 'white')]
```

Now we iterate on this, so we get the different tuples one after the other.

From these tuples we print c[0], the index and c[1], the colour text, separated by a tab.

So as a result we get:

```
0       black
1       brown
2       red
...
8       grey
9       white
```

19. Functions

It is a good idea to put pieces of code that do a clearly defined task into separate blocks that are called functions. Functions must be defined before they are used.

Once they are defined, they can be used like native Python statements.

A very simple example calculates area and perimeter of a rectangle:

```
# function definitions
def area(b, h):
    """ calculate area of a rectangle"""
    A = b * h
    return A

def perimeter(b, h):
    """ calulates perimeter of a rectangle"""
    P = 2 * (b+h)
    return P

# main program using defined functions
width = 5
height = 3
print "Area = ", area(width, height)
print "Perimeter = ", perimeter(width, height)
```

The syntax of a function definition is:

```
def <function_name(<argument1>, <argument2>, ....):
        <statements>
        ....
            return <returnvalue(s)>
```

The arguments are the values passed to the function.

the return value is the value that the function gives back to the calling program statement.

Dont't forget the ":" and the indentation !

A function can return more than one value:

```
# function definition
def area_and_perimeter (b, h): A = b *
    h
      P = 2 * (b+h)
       return A, P

# main program using defined function ar, per =
area_and_perimeter ( 4, 3) print ar
print per
```

Here the return values are returned as a tuple.

If the function doesn't need to return a value, the return statement can simply be omitted.

Example:

```
# function definition
def greeting():
        print "HELLO"


# main program using defined functions
greeting()
```

One good thing about functions is that they can be easily reused in another program.

Notes:

- Functions that are used often can be placed in a separate file called a module. Once this module is imported the functions can be used in the program.
- It is possible to pass a variable number of arguments to a function.

For details see here: http://en.wikibooks.org/wiki/Python_Programming/Functions

- It is possible to pass named variables to a function

20. Avoiding for loops: vector functions

For loops tend to get slow if there are many iterations to do.

They are not necessary for calculations on numbers, if the **Numpy** module is used. It can be found here http://www.numpy.org/ and must be installed before using it.

In this example we get 100 values of a sine function in one line of code:

```
import numpy as np

# calculate 100 values for x and y without a for loop
x = np.linspace(0, 2* np.pi, 100)
y = np.sin(x)

print x
print y
```

21. Diagrams

Once you have calculated the many function values, it would be nice to display them in a diagram. This is very simple if you use Matplotlib, the standard Python plotting library.

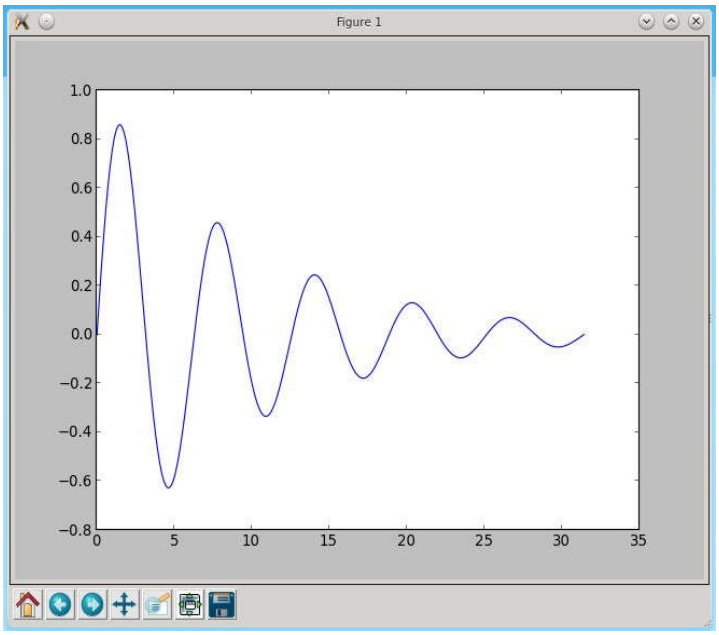
Matplotlib can be found here: http://matplotlib.org/downloads

The following program calculates the values of a function and draws a diagram:

```
from numpy import linspace, sin, exp, pi import
matplotlib.pyplot as mp

# calculate 500 values for x and y without a for loop
x = linspace(0, 10*pi, 500)
y = sin(x) * exp(-x/10)

# make diagram
mp.plot(x,y)
mp.show()
```



Notes:

- Matplotlib offers much more than this, see online documentation.
- There are two ways to use Matplotlib: a simple functional way that we have just used, and a more complicated object-oriented way, that allows for example to embed a diagram into a GUI.

22. Appendix

Importing functions from a module

Three ways to import functions:

1.

the simplest way: import everything from a module

advantage: simple usage e.g. of math functions

disadvantage: risk of naming conflicts when a variable has the same name as a module function

```
from numpy import *
print sin(pi/4)

# With this import method the following would give an error:
#sin = 5                    # naming conflict!
#print sin(pi/4)
```

2.

import module under an alias name that is short enough to enhance code clarity

advantage: it is clear to see which function belongs to which module

```
import numpy as np print
np.sin(np.pi/4)
```

3.

import only the functions that are needed

advantage: simple usage e.g. of math functions
naming conflict possible, but less probable than with 1.

disadvantage: you must keep track of all the used functions and adapt the import statement if a new function is used

```
from numpy import linspace, sin, exp, pi print sin(pi/4)
```