

Introduction to Python

- Master 1 SIED -



Courses & tutorials: <http://bioinfomed.fr/> => teachings

Contacts: croce@unice.fr

Why Python?

To program, we need a language. There are many of them, each being more or less easy to tackle, each more or less adapted to a particular field. Python has many advantages:



- The syntax of the language encourages **clarity**, which greatly facilitates the rereading of programs it is indeed very painful to "repeat" a program of someone else, or that one wrote oneself long before, whose simple visual aspect is confused
- It is an **object-oriented language**, currently the most widespread programming paradigm. However, it also could be used as **procedural language** (most of this course...)
- It is an **interpreted language**, so its implementation is very simple
The interpreter, the software that allows your programs to run, is **free** and **multi-platforms** software: a program written under Linux can run under Windows without any modification
- **Modern language** and very **fast development**
- It is quite pleasant because of "**high level**".

Why Python?

It is necessary to install the Python interpreter on your computer. The procedure to follow differs according to your operating system

Under Linux or Mac it is often installed by default.

Under Windows the installation is very easy (consult the Python website to download the installation program “www.python.org”)

It is recommended to install TKinter (graphics library) to take full advantage of Python.
(www.activestate.com)

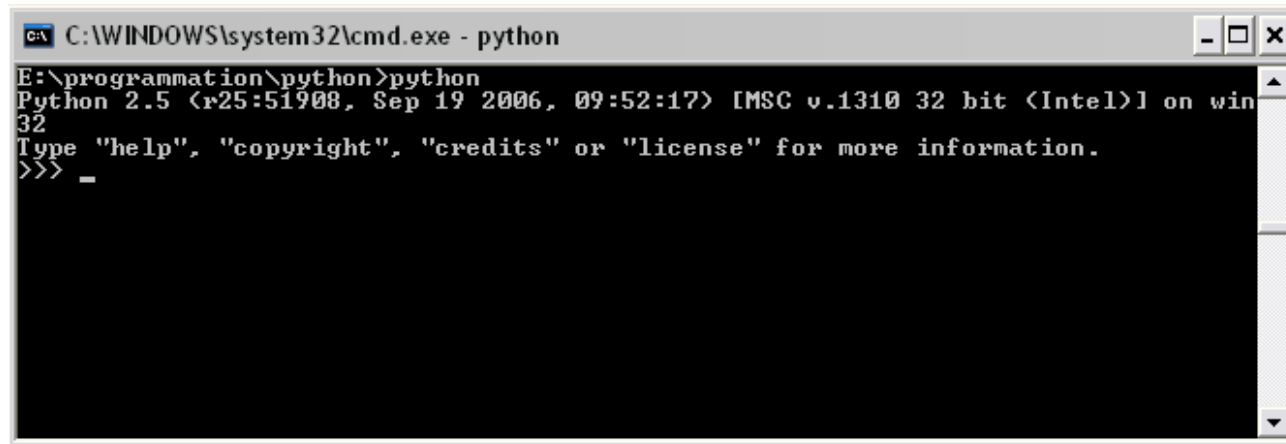
2 different versions of Python exist : V2 and V3

=> Prefer Python V2 (ig. Python 2.7) for these courses. Now, install Python on you computeur !

First instructions with Python

The heart of interpreted language is the “interpreter”.

To run it, open a terminal ("cmd" under windows), then run the python command. Note: if the command is not in the "path" it must be run from the installation location

A screenshot of a Windows command prompt window. The title bar reads "C:\WINDOWS\system32\cmd.exe - python". The command prompt shows the user typing "python" at the prompt "E:\programmation\python>". The output is "Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32". Below this, it says "Type 'help', 'copyright', 'credits' or 'license' for more information." and then shows the prompt ">>> _".

```
C:\WINDOWS\system32\cmd.exe - python
E:\programmation\python>python
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

The characters >>> on the left are the prompt of the Python interpreter, which tells you that it is ready to receive your instructions.

Try (pressing Enter at the end of each line):

2+3

5.95 * 6.55957

print "Hello!"

First instructions with Python

These simple examples already give us some information:

- you can use the Python interpreter as a calculator
- to make an addition, it is written naturally, using the operator +
- decimal numbers are written in English, using a point (.) instead of a comma (,)
- the multiplication is done by the operator * (the star, or asterisk)

Note that spaces between the values and the operator can be inserted or not
the strings, are delimited by the character " (or double-quote)

x = 5 <= here your first variable

To reuse this variable, simply enter its name, for example :

2*x

This gives: 10

we can redefine the variable:

x = 6

x = 2*x

This gives: 12

First instructions with Python

Variables can naturally be used to store things other than integers, and names are not limited to a letter :

```
pi = 3.14159
```

```
master = "M2 SIED"
```

In Python language, as in most languages, variable names can be composed of letters of the alphabet, numbers and underscore (`_`) characters, with the constraint that the first character must not be a number.

In addition, Python is **case sensitive**: `x` and `X` can be two different variable names. In general, it is strongly recommended to give explicit names to variables.

In the previous examples, `x` is an **integer** type variable, `pi` is a “real” or a “**float**” (=floating point) , `master` is a **string** (=str) type variable. Some languages, like Python, determine the type of a variable when it is declared (which also happens to be an assignment). Others, such as C/C++, require this type to be explicitly defined before any assignment.

Make a source code

Using the interpreter as we have done so far does not make it possible to build a complex program, simply because the instructions would always have to be retyped. This is why the instructions of a program are stored in one or several files, all these files constituting the source code: the code typed by the programmer to obtain the desired program.

To create **source code**, all you need is a text editor. Attention, it is indeed plain text, without any formatting so a simple editor is sufficient.

Scite (= www.scintilla.org) is available under Windows and Linux. It is simple and very well suited for coding in Python or another language. (other editors (=IDE), **Spyder**, **Eric4**, **Bluefish**, ...)

Open Scite (or another text editor you prefer) and type:

```
print "Hello !"  
str = "x is"  
x = 5  
print str, x
```

save your file with the “.py” extension

Make a source code

Your program must be interpreted to run:

To interpret it, from a command line, run "**python prog.py**".

Installing python under Windows usually associates “.py” files with the python interpreter. Double-click directly on the file in the explorer.

With **Scite** it is possible to start directly the interpretation of the current script by pressing **F5**. The result is then displayed in a pop-up window.

Handling of variables (part 1)

Generality on variables

A variable has a **type**, which specifies the nature of the information it may contain.

Test it:

```
x = 1  
print type(x)
```

type() is a **function** (collects instructions). Here the role of this function is to give the type of what is between the parentheses (we also say, its **parameter**).

Test it:

```
x = 3.14  
print type(x)
```

Handling of variables (part 1)

Conversions between types

Performing a type conversion consists in consulting the contents of a variable as if it were of a different type. For example, let's create a string variable :

```
s = "10"
```

This string represents a numerical value, but for the moment it is only a string. Add a value:

```
print s + 5
```

 What happens?

The solution is to perform a conversion, in our case to transform the character string into a corresponding numerical value :

```
print int(s) + 5
```

int() is a function that tries to make an integer from what is given in parentheses

Test it:

```
s = "error !"
```

```
print int(s) + 5
```

Then:

```
print int(10.8) + 5
```

Handling of variables (part 1)

Test:

```
s = "10.7"
```

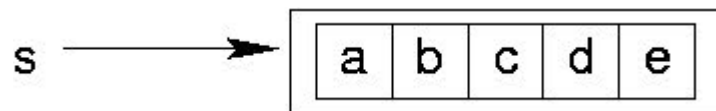
```
x = float(s) + 1.1
```

```
print x
```

```
print str(x) + "0"    Result?
```

Strings of characters

Strings are actually a series of characters in memory, which can be represented in the form of a table



It is possible to obtain an individual element from this field, i.e. an individual character:

```
print s[2]
```

c

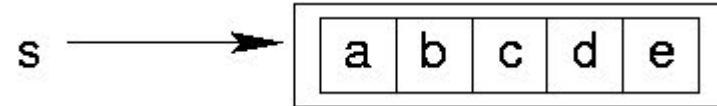
Handling of variables (part 1)

The string is considered as a table. Using the brackets allows us to indicate which element of the table we want to obtain. The number given in brackets is called the **index** of the desired **element**. Here, we ask for element at index 2. By convention, **index start at 0**

It is possible to give **negative index**, the numbering then begins with the end :

```
print s[-2]
```

```
d
```



Cutting

```
print s[1:4]
```

```
bcd
```

```
print s[1:-2]
```

```
bc
```

If the first member is missing, it means "from the beginning". If the second is missing, it means "to the end". For example :

```
print s[2:]
```

```
cde
```

Handling of variables (part 1)

Assemblage (=concaténation »)

```
s1 = "hello"
```

```
s2 = "world !"
```

```
s = s1 + s2
```

```
print s
```

The string `s` is the result of the concatenation of the string `s1` and `s2`. In Python, concatenation is performed by the `+` addition operator (if the members are strings !)

```
s1 += s2
```

```
print s1
```

Hello, world!

Combines in a single operation the concatenation then the assignment in the given string to its left. The line `s1 += s2` is therefore equivalent to `s1 = s1 + s2`, i.e., "paste the strings `s1` and `s2`, then store the result in `s1`

Handling of variables (part 1)

Exercise:

Let the sentence "Hello world! »

Make a script that allows to insert the string "all" in s between "Hello" and "world".

- display s

- Correct "all" to "all the" and display the sentence again

Another trick on strings: it is possible to repeat a string by multiplying the string by an integer

Example:

```
s = "toto" * 5
```

```
print s
```

```
toto toto toto toto toto toto
```

Handling of variables (part 1)

Arithmetic operations

```
a = 5
a += 2
print a
7
```

```
2**128
340282366920938463463374607431768211456
2.0**128
3.4028236692093846e+38
```

Mathematically, "2" and "2.0" are perfectly equivalent, but the computer does not store them at all in the same way. Python returns an exact result in the first case (39 digits), but an approximate result in the second ("only" 17 digits).

Decimal numbers have limited accuracy. In general, as soon as the writing of the number requires more than 16 digits, the computer makes a rounding and manipulates only an approximate value, from where a certain risk of error for important calculations

```
10.0**50 + 2 - 10.0**50
0.0          Warning !!!
```

=> *See tutorial*

The functions

The functions are mini-programs that avoid repeating a sequence of instructions.

A function is therefore a sequence of instructions that can be executed on demand by a program: everything then happens as if the instructions of the function had been inserted at the place where it is called.

Define a function

```
def my_fonction () :
```

```
    instruction1
```

```
    instruction2
```

```
instruction 1, main program
```

```
My_function()
```

```
instruction 3, main program
```

```
...
```

- the “**def**” keyword: we inform Python that we are going to define a function
- then the name of the function, here "my_function”
- then a pair of parentheses “()” ; they are necessary, they possibly contain arguments
- The colon “:”, which tells Python that the following instructions (with **indentation**) will be contained in the function

The functions

Instructions contained, for example in a function, are a **block of instructions**

In python a block is defined by a similar **indentation** for a sequence of instructions (tabs, spaces, etc.).

Depending on the languages, different techniques are used to locate the beginning and end of a block: for example, in Ada or Pascal languages, the beginning is marked by begin and the end by end. In C/C++, php, perl we use { and } braces

At the end of each instruction line, there is no character, unlike other languages where each instruction ends with an ;

Empty lines are not interpreted and are simply used to make the code more readable

The functions

Call a function

To call a function, simply enter its name, with a pair of parentheses containing arguments if they exist. In addition, the function must be defined before it is called.

Exercise: repeat the following code using a function

```
print "line 1"  
print "_"*10  
print "-"*10  
print "line 2"  
print "_"*10  
print "-"*10  
print "line 3"  
print "_"*10  
print "-"*10  
print "line 4"
```

The functions

Exercise (correction):

```
def line():
```

```
    print "_"*10
```

```
    print "-"*10
```

```
print "line 1"
```

```
line()
```

```
print "line 2"
```

```
line()
```

```
print "line 3"
```

```
line()
```

```
print "line 4"
```

The functions

Function parameters (=arguments)

The parameters of a function make it more generic, i.e. it can adapt its action according to the situation.

The parameters are transmitted to the function during a call. During the call they can be variables or directly values. If it is a variable, Python replaces it with its value.

At the function level the value(s) are sent in variables (to be named in parentheses)

```
def My_function (parameter) :
```

```
    x = 1
```

```
    instruction2
```

<= beginning of the main program

```
instruction 1
```

```
My_function(value)
```

```
instruction 3
```

Variables in a function are "local" i.e. they exist only in the function, the main program does not know them.

The functions

It is possible to give several parameters to a function.

The function parameters are retrieved in the same order as during the call.

Example:

```
def Ma_function (param1, param2) :
```

```
    instruction1
```

```
    instruction2
```

```
    param1 = value1
```

```
    param2 = value2
```

```
instruction 1
```

```
Ma_function(value1, value2)
```

It is however possible to give arguments in the wrong order if the values passed during the call are "named".

```
def Ma_function (param1, param2) :
```

```
    instruction1
```

```
    instruction2
```

```
instructions 1
```

```
Ma_function(param2=value2, param1=value1)
```

The functions

It is possible to specify, in the function definition, a default value for one or more parameters, i.e. the value they will have if they are not present when the function is called.

```
def Ma_function (param1="toto", param2="tata") :
```

```
    instruction1
```

```
    instruction2
```

```
param1 = value1
```

```
param2 = "tata"
```

instruction 1

Ma_function(param1=value1)

On the other hand, if you do not specify a default value and you put fewer parameters when calling than in the function, there will be an error.

The functions

Exercise:

Modify the previous program so that you can specify when calling the function:

- the character to be displayed (characters `_` and `-` in the current program). You can put the characters you want...
- the number of times the character must be repeated (in the current program, the characters are repeated 10 times) Call the function with the value 10, then 20, then 30.

def line():

```
    print "_"*10
```

```
    print "-"*10
```

```
print "line 1"
```

```
line()
```

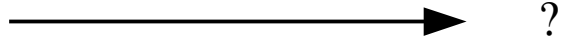
```
print "line 2"
```

```
line()
```

```
print "line 3"
```

```
line()
```

```
print "line 4"
```



?

The functions

Exercise (possible correction)

```
def line(car="-", nb=10):  
    print car*nb
```

```
print "line1"  
line()  
line("$",10)  
print "line2"  
line("-",20)  
line("*",20)  
print "line3"  
line("-",30)  
line("@",30)  
print "line4"
```



```
line1  
-----  
$$$$$$$$$$$$  
line2  
-----  
*****  
line3  
-----  
@@@@@@@@@@@@@@@@@@@@@@@@  
@@@@@@@@@@@@  
line4
```


The functions

Returns of functions

The ability to define functions is a major principle of structured programming. Efforts are made to organize programs in a way that divides larger problems into smaller ones. This creates program elements that communicate and work together, while avoiding as much as possible that these elements depend too much on each other.

The objective being to avoid the mess that could be old programs written for example in Basic language.

In the previous program we have a function that performs a calculation, namely build a string and display it. However, in a "good" program, a function does not have to "know" what to do with this string, it just has to do the calculation.

For example, using this function, how is it possible to write this output into a file instead of displaying this string. It would be necessary to modify the function, it is what it is necessary to avoid...

The functions

So the idea is to define functions that do what they were created for, but no more. Our function calculates a string but it is not its role to display it. In our case, this role is assigned to the main program. We therefore need a way to "recover" what the function has produced. We then say that the function will return a result, this result being the return value of the function. What we will do with this result is the responsibility of the one who calls the function, in our case the main program.

Example:

```
def ma_function (var1, var2) :  
    result = var1 + var2  
    return result
```

```
my_variable = my_function(2,3)  
print my_variable
```

- “result” is a **local variable** (internal to the function and unknown by the main program)

- **return result** means that the function returns the content of the variable “result” to the main program

- In the main program the return value of the function is here stored in the variable "my_variable"

=> See tutorial

Interactions with a program

1 - Creating a simple game: interacting with the user

Controlling or interacting with the progress of a program generally means directing the sequence of instructions, for example choosing between certain sequences according to various criteria, or causing the execution of a certain sequence to be repeated.

To illustrate these principles, we will make the following game: a program must choose a random number between 1 and 10, and the user will have to find this number by making proposals (keyboard input). For each proposal, the program will display "too large" or "too small", as appropriate.

```
from random import randint
```

```
random_number = randint(1, 10)
```

```
string = raw_input("Give a number: ")
```

```
player_number = int(string)
```

1- This line gives access to a function named `randint()`, which allows to obtain a random number between two numbers given as parameters. It is contained in a "module" (=set of similar functions grouped together)

2 - The result of the `randint()` function is stored in the `random_number` variable. It is this number that the user will have to find by successive tests.

3 - The `raw_input()` function is the opposite of the `print` command, which we have already used a lot. Its role is to wait for the user to enter something on the keyboard, the string that is passed as parameter being displayed at the beginning of the line.

4 - `raw_input()` always returns a string, and we want a number. It is therefore necessary to transform this string into a number, which will be stored in "`player_number`".

Interactions with a program

2 - Creating a simple game: from the algorithm to the program

- After the user has entered his number it is necessary to display a response that takes into account the number of the user and the random number created by our program. There will therefore be a "comparison" of variables.

By forgetting for the moment the case where the number the user finds exactly the random number, our algorithm can be broken down as follows:

if the given number is greater than the reference number, then the message is "Too big", otherwise the message is "Too small".

- The goal is to approach the Python language, replacing some terms by the variables they represent:

*if **player_number** is greater than **random_number**, then display etc....*

The notion of message, in our case, corresponds to a display on the screen, which the print command allows. In addition, two numbers are compared using the usual mathematical operators < and >

Interactions with a program

if number_player > random_number, then print "Too big", otherwise print "Too small"

In python the "if" is represented by "if"

The "then" is represented by the ":"

“Otherwise” is represented by "else"

If (condition) : (instructions to be executed)

else : (instructions to execute)

Taking our previous algorithm, this gives:

```
if ( number_player > random_number) . :  
    print "Too big!"  
else :  
    print "Too small!"
```

Note that instructions are indented to their condition.

The parentheses for the “if” are optional but it is better to put them (increase visibility).

The “else” is obviously not mandatory if there is only one possible alternative.

Now modify the program to take into account the case of equality between the 2 numbers.

The comparison operator for testing equality is “==” (double equal)

Interactions with a program

```
if ( player_number == random_number ) :  
    print "You found it!"  
else :  
    if ( number_play > random_number ) :  
        print "Too big!"  
    else :  
        print "Too small!"
```

There is an **association of blocks** here.

Our program is not finished: currently the user can only propose one number, then the program ends. How can the user propose several numbers? One solution would be to duplicate the lines of code. But that doesn't seem like a good solution, especially if you're faced with hundreds of instructions rather than a few.

Now, we want the user to try several values to find the random number

Interactions with a program

3 - Creating a simple game: introduction to repetitive structures (=loops)

until the numbers are equal, display a suitable message then ask for a new number.

The "display a suitable message" part has already been processed, so we can replace this member with our previous principle (which we already know how to translate):

*As long as the numbers are not equal, if the given number is greater than the reference number, then display "**Too large**", otherwise display "**Too small**", then ask for a new number.*

The word "as long as" implies that we will perform the following actions until a certain condition is verified, or more precisely here, until a condition is verified. First problem, how to translate the expression "are not equal"? It is enough to express the negation of the condition "are equal". In Python language, the negation of a condition is simply written "**not**". By replacing with the variables we have, and using the comparison operators, we can rewrite the sentence as follows:

*as long as **not** (number_player == random_number), if (number_player > random_number) then print "Too big", then print "Too small", then request a new number.*

Considering that « as long as » is translated in Python by « while », complet the program...

Interactions with a program

Full program:

```
from random import randint
random_number = randint(1, 10)
string = raw_input('Give a number : ')
number_player = int(string)

while ( not (player_number == random_number)) :

    if ( number_player > random_number) . :
        print "Too big!"
    else :
        print "Too small!"

    string = raw_input("Give a number :")
    number_player = int(string)

print "bravo !"
```

Note: The loop condition can also be written this way:

```
while (number_player != random_number) :
```

The operator "!=" means "is different".

Interactions with a program

3 notions to remember:

- The notion of **alternative**, which makes it possible to direct the flow of a program. As a result, certain parts of a program will only be executed under certain specified conditions.
- The notion of **loop**, which allows to execute several times the same sequence of instructions. The end of the loop depends on a **given condition**.
- The notion of **condition** that is implicit in the first 2. The value of a condition is given by the result of a particular calculation, which is the test corresponding to the condition. Strictly speaking, the type of this value is neither an integer nor a string, it is a type called **Boolean** (true or false, 0 or 1)

These three notions are part of the foundations of **algorithmic**. As a reminder, an algorithm is the description of a series of actions or operations to be performed in a certain order. During this sequence, loops or alternatives can act.

Summing-up exercise: repeat the last program and modify it as follows:

- Create a function that asks the user for the number and returns this number to the main program (the function is called twice in the main program and replaces 4 instructions)
- After finding the number, the program must propose to the user to replay:

If the user types "yes", the game is restarted, if "no" or something else, the program ends.

Interactions with a program

```
from random import randint
```

```
def number ():
```

```
    string = raw_input("Give a number :")
```

```
    string = int(string)
```

```
    return string
```

```
play = "yes"
```

```
while (play == "yes") :
```

```
    random_number = randint(1, 10)
```

```
    number_player = number()
```

```
    while (number_player != random_number) :
```

```
        if ( number_player > random_number) . :
```

```
            print "Too big!"
```

```
        else :
```

```
            print "Too small!"
```

```
        number_player = number()
```

```
play= raw_input("Bravo! Do you want to play again?")
```

Handling variables (part 2)

Reminder on variables:

- simple variables: integers, float, etc.
- the strings

	0	1	2	3							-3	-2	-1
	m	y	_	e	x	a	m	p	l	e	.	p	y
	:												:

the lists

The list is the most flexible ordered collection item. It can contain all kinds of objects: numbers, strings, and even other lists. The declaration of a list is made by placing the objects of the list in square brackets.

```
x =[0, 1, "hello"]
```

From the point of view of usage, lists make it possible to collect new objects in order to process them all together.

Like strings, objects in a list are selectable by specifying them. The lists are ordered and therefore allow for slice extraction and concatenation.

The lists allow a modification of their contents: replacement, destruction,... of objects without having to make a copy (like strings). They can be modified on the spot.

Handling variables (part 2)

Operation	Interpretation
<code>L = []</code>	Empty list
<code>L = [0, 1, 2, 3]</code>	4 index items from 0 to 3
<code>L = ['abc', ['def', 'ghi']]</code>	included list
<code>L[2]</code>	element at index 2 (3d element)
<code>L[2:5]</code>	all elements between indexes 2 and 5
<code>len(L)</code>	length of list L
<code>L1 + L2</code>	concatenation of these 2 list
<code>L1 * 3</code>	multiplication of list L1
<code>for x in L</code>	loop on items of the list (=iteration)
<code>3 in L</code>	is value 3 is contained in list L
<code>L.append(4)</code>	add '4' in list L
<code>L.sort()</code>	sort L (alphabetical or expanding)
<code>L.index()</code>	search
<code>L.reverse()</code>	reverse items of L
<code>del L[3]</code>	delete items in index 3 of list L
<code>L[2] = 12</code>	put value 12 in list L at the index 2
<code>L[3:5] = [12,45,23]</code>	put 3 values for the 3 sliced index

Nested lists are **n-dimensional lists**.

These are **matrix**.

Example:

```
list1 = ['toto', 'tata']
```

```
list2 = ['titi', 'tutu', list1]
```

```
print list2[2][0]
```

--> toto

Handling variables (part 2)

The Dictionaries

The dictionary or hash table is a very flexible system for storing multiple values. A dictionary can be compared to a list, but instead of the indexes (0,1,2,3,..), it contains a "word" called a "**key**" that makes it easy to find the corresponding **value**.

A dictionary is assigned by a pair of braces {}.

Example:

```
dictio = {'japan' : 'japon', 'china' : 'chine'}
```

```
print dictio['china']
```

```
chine
```

Attention the use of dictionaries (as when displaying) is the same as lists: with []

Keys are usually **strings**, but they can also be other types.

The stored values can be of any type: **strings, numerical values, lists, dictionaries,...**

The keys of the dictionary are **not ordered** contrary to the indexes of the lists.

Handling variables (part 2)

Operation	Interpretation
<code>d = {}</code>	Empty dictionary
<code>d = {'one' : 1, 'two' : 2 }</code>	Dictionary with 2 elements
<code>new = {'count': {'one': 1, 'two' : 2}}</code>	A dictionary as a value of the first key of the dictionary 'new'
<code>d['one'] = 10</code>	Add (or modify) with the value 10 at the key 'one' of the dictionary 'd'
<code>new['count']['two'] = 20</code>	Add 20 at the key 'two' of the dictionary included at the key 'count' of dictionary new
<code>d.has_keys('one')</code>	return true (or 1) if the value 'one' exists in d
<code>d.keys()</code>	return a list containing all the keys of the dictionary d
<code>d.value()</code>	return a list containing all the values of the dictionary d
<code>d.len()</code>	return the length of the dictionary d
<code>del d['one']</code>	delete the keys 'one' (and the corresponding value) for the dictionary d

Handling variables (part 2)

Les tuples

The tuple is poorly used in Python. It's like the list, but the difference is that the values put in a tuple are no longer editable. Once created, you cannot add or delete elements

A tuple is declared with values in parentheses and not in square brackets like the list.

```
tuple = (0, 1.4, "world")
```

```
print tuple[1]
```

1.4

Operation	Interpretation
T = ()	Empty tuple
T = (0, 1, 'hello')	create a tuple with 3 elements
T = (0, 1, ('hi', 'world'))	create a tuple with 3 elements. The 3d element is a tuple of 2 elements (with 2 strings)
T[2]	element at index 2
T[2:5]	all elements between indexes 2 and 5
len(T)	Length of the tuple T
T1 + T2	concatenation of 2 tuples
T1 * 3	multiplication of tuple T1
for x in L	loop on items of the tuple T (=iteration)
3 in T	is value 3 is contained in tuple T

Handling variables (part 2)

Global variables

By default, variables contained in a function are not visible outside the function.

A variable declared as "**global**" becomes visible everywhere in the program

```
def my_func():
```

```
    global x          # Global value
    y = 12            # Local value
    return x*y
```

```
x = 'hello'
```

```
print ma_fonc()
```

```
hello
```

=> *See tutorial*

The exceptions

Exceptions are a way to "work around" a problem that can create a bug

Exemple:

try:

```
x = int(raw_input("Please enter a number:"))
```

except :

```
print "Aille! It was not a valid number. Try again..."
```

Exceptions are not "conditions" (if, else,...)

Iterations (=loops)

Loops “while”

The Python statement "**while**" is the most common iteration construct.

"while" consists of a header line with a test expression, followed by a block of several instructions.

"while" repeatedly executes the indented block as long as the condition test is performed. If the condition is immediately false the block will never be executed.

There may be an optional "**else**" part, which is executed if the control exits the loop without using the break instruction. (different from "else" after "if")

```
while <test> :  
    <instructions>  
else :  
    <instructions>
```

Example :

```
i=0  
while i<3 :  
    print “hello”  
    i =i+1
```

hello

hello

hello

Iterations (=loops)

Loops “for”

The "for" loop allows to obtain all the values contained in a variable like lists, tuples or dictionaries and even strings (the elements will then be each character of the string).

Each loop corresponds to one of the values contained in the variable. The value is then stored in another variable for the loop time:

```
for <temporarily_variable> in <variable> :  
    <instructions>
```

Example:

```
a = ['cat','dog','pig']           # a list with 3 values  
for x in a:                   # initiation of a loop "for" using variable a. Values will be store in x  
    print x                     # print the variable "x"
```

cat

dog

pig

Note: the "for" loop in Python works differently than in C !

Iterations (=loops)

“break”, “continue”, “pass” and the “else” of loops

These three instructions, "pass", "break", and "continue", are used to manage the continuity of a loop according to the actions that take place inside it.

- The "break" instruction is intended to exit the loop instantly and move on to the next step
- "continuous" jumps directly to the beginning of the next loop.
- "pass" does nothing at all, but since you can't have an expression that isn't followed, "pass" can be used to fill that void.
- the "else" at the end of the loop is different from the "else" with the "if". The block that follows this "else" is executed if the loop normally ends, i. e. without "break" or another output before the end.

```
for <cible> in <objet> :  
    if <test1>: break  
    elif <test2>: continue  
    else: <instructions>  
else :  
    <instructions>
```

Iterations (=loops)

Use of « for », « range() » and « len() »

The "for" does not need a counter variable (ie. $i=i+1$) as the "while". 2 solutions if you need a counter with the "for":

- add a counter! (ie. $i=0$ and $i=i+1$ inside the loop)

- use the function "**range()**" + "**len()**"

- len() : gives the number of elements contained in a variable (string, lists, dictionaries, etc)

- range() : automatically creates a list including values 0,1,2,3,4,...

"range()" can have 1, 2 or 3 arguments.

```
x = range(10)           x -->    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
y = range(10,20)       y -->    [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
z = range(10,20,3)     z -->    [10, 13, 16, 19]
```

```
a = ['Marie', 'had', 'one', 'small', 'sheep']
```

```
max = len(a)
```

```
for i in range(max):
```

```
    print i, a[i]
```

0 Marie

1 had

2 one

3 small

4 sheep

=> *See tutorial*

The files

Python is able to easily read or modify any text files.

It first needs to “open” the file, using function “**open()**”

2 arguments are required :

1) the name of the file with the path (or without if the file is present at the same location of the script)
File name must be the exact name, including eventual extension like “.txt” (could be hidden under Windows)

2) the mode to open this file :

“**r**” : ready only mode

“**w**” : write only mode. The previous content of the file will be erase if the file existed

“**a**”(=append)“. Write only, the content will be kept: additionnal text will be added at the end

“**r+**”. Read and write are allowed

```
my_file = open(«my_document.txt», «w»)
```

Here the file “my_document.txt” is open in write mode. “my_file” is now the variable (=> object) to be used to manipulated the content

The files

Main functions (methods) to be used with files :

Operation	Interpretation
<code>out = open('tmp/spam.txt', 'w')</code>	Create (write) an output file named here "spam.txt"
<code>in = open('data', 'r')</code>	Open a file "data" to be read
<code>s = in.read()</code>	Read all the content of the file "in" in a string
<code>s = in.read(N)</code>	Read until N octets (=characters)the content of the file "in" in a string
<code>s = in.readline()</code>	Read a line of the file "in", in a string. Then, move the cursor to th next line
<code>L = in.readlines()</code>	Read all the file in a list. Each line is an element of the list
<code>out.write(s)</code>	Write the string s in the file out
<code>out.writelines(L)</code>	Write the content of list L in the file out. Each element is a line
<code>out.close()</code>	Close the file.

The "close()" method closes the current file object. Python, to manage memory space, closes the file itself in some cases when it is no longer referenced anywhere. However, closing a file manually allows some clarity, which is important in a large application.

The files

Exercise:

- create a list containing 5 values: "line1", "line2", "line3", "line4", "line5".
- browse the list with a "for" to modify each value by adding the character "\n" (= return to the line)
- display all the items in the list (with a "for")

The result should look like this:

line1

line2

line3

line4

line5

The files

Exercise (continued):

- create a file 'file.txt' in the current location of your python script (".file.txt") ('w' mode)
- write in this file the content of the previous list
- close the file

note: methods to be used: open, writelines, close

We want to add a line at the end.

- Open the file and write this string: "last line" without deleting the previous content ("a" mode)
- close the file

note: methods to be used: open, write, close

- display the contents of the file ('r' mode opening)
- close the file

note: methods to be used: open, read, close

The files

Exercice (correction):

```
list = ['line1','line2','line3','line4','line5']
```

```
for i in range(len(list)):
```

```
    list[i]=list[i)+"\n"  
    print list[i]
```

```
myfile = open('file.txt','w')  
myfile.writelines(list)  
myfile.close()
```

```
myfile = open('file.txt','a')  
myfile.write("last line")  
myfile.close()
```

```
myfile = open('file.txt','r')  
content = myfile.read()  
myfile.close()
```

```
print content
```

=> *See tutorial*