

Introduction to Python



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**".

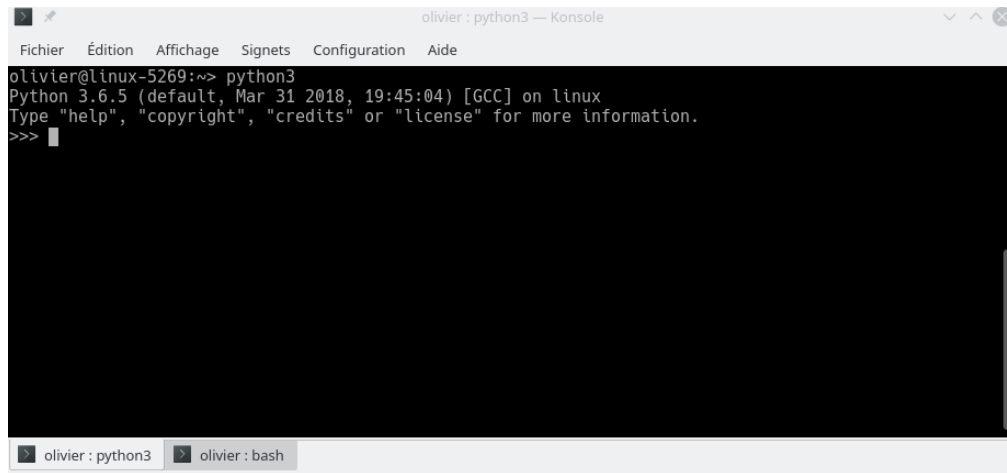
Why Python?

- It is required to have the “Python interpreter” (a single program) installed on your computer.
- Under Linux or Mac it is often installed by default. To check if Python is already installed, open a console and type “python --version”
- Under Windows you need to install an interpreter of Python.
See www.python.org to download Python v3.x. Note at the beginning of the installation you need to check the box “add python to environmental variable” (otherwise, you need to add “python to the path” later, see google for more info)
- “Tkinter” is an additional library for python for building graphical interfaces. This lib is installed by default with recent versions of python, but if not, it is recommended to installed it (no need for the first part of this teaching)
- There are 2 different versions of Python : v2 and v3. Only few differences, but they are not fully compatible
=> Prefer Python v3 for these courses. Now, install Python on you computer !

First instructions with Python

The heart of interpreted language is the “interpreter”.

To run it, open a terminal ("cmd" under windows or a console under Mac or Linux), then run the python command.



```
olivier@linux-5269:~> python3
Python 3.6.5 (default, Mar 31 2018, 19:45:04) [GCC] on linux
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 " (quotes 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

print (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 = "Master at UCA"
```

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 enough.

www.sublimetext.com (win, mac, linux)

thonny.org/ (win, mac, linux)

code.visualstudio.com/ (win, Mac, linux)

www.jdoodle.com/python-programming-online (no installation, online interpreter by your browser)

developer.apple.com/xcode/ (mac)

www.jetbrains.com/fr-fr/pycharm (mac, linux, windows)

www.scintilla.org/wscite420.zip (windows and linux)

<https://jupyter.org/> (online interpreter)

Spyder, Eric4, Bluefish, sublimeText, Atom IDE, ... (more here :

<https://wiki.python.org/moin/PythonEditors>) => **install a Python editor on your computer**

Make a source code

Open your python editor, create a new file, type some python commands inside and save your file with a name ended with “**.py**” extension. Example :

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

Your program must now to be interpreted. 2 solutions :

- Editors usually propose a button in the menu or a keyboard shortcut to execute (=interpret) your code. Actually, the editor use your existing python installation.
- You can also, go the you console, where is script is (“cd” command, to change the folder) and directly interpret the code that is inside your file by typing for ig. "**python your_prog.py**".

Note, when using an editor, it does exactly the same command, but automatically.

Remark : put “#” as first character of a given line, to avoid interpreting this line = a line of comments

#print (str, x) => this is an inactive line into your code (a comment)

blabla, this is a comment

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
```

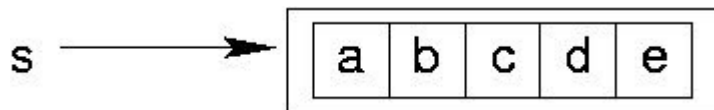
```
print (x)
```

```
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:

```
s = "abcde"
```

```
print (s[2])
```

=> c

c

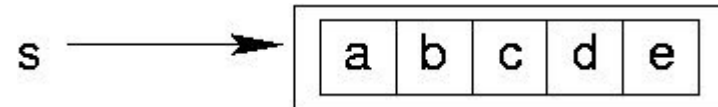
Handling of variables (part 1)

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":

```
print (s[:2])          => ab
```

If the second is missing, it means "to the end":

```
print (s[2:])          => cde
```

Handling of variables (part 1)

Assembly (=concatenation »)

```
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
```

<=>

```
s1 = 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! » in s

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

```
s = "Hello world!"
```

```
s = ...to complete.....
```

```
print (s)
```

=> should be : Hello all world!

```
s = ...to complete.....
```

```
print (s)
```

=> should be : Hello all the world!

Another trick about 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

Handling of variables (part 1)

Arithmetic operations

<code>a = 5</code>	<code>2**128</code>
<code>a += 2</code>	<code>340282366920938463463374607431768211456</code>
<code>print (a)</code>	<code>2.0**128</code>
<code>7</code>	<code>3.4028236692093846e+38</code>

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 1 (ex1 & ex2)*

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_function () :
```

```
    instruction1
```

```
    instruction2
```

example of an instruction : `print ("my text")`

```
instruction x
```

```
my_function()
```

```
instruction y
```

```
...
```

- 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) :  
    parameter = 1  
    instruction2
```

instruction 1

my_function(value)

instruction 3

1st line of the main program

value could be anything : integer, text, string, ...

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 my_function (param1, param2) :  
    instruction1  
    instruction2  
  
param1 = value1  
param2 = value2
```

```
instruction 1  
my_function(value1, value2)
```

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

```
def my_function (param1, param2) :  
    instruction1  
    instruction2  
  
instructions 1  
my_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 my_function (param1="toto", param2="tata") :
```

```
    instruction1
```

```
    instruction2
```

```
param1 = value1
```

```
param2 = "tata"
```

```
instruction 1
```

```
my_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()

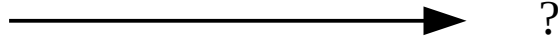
print ("line 2")

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("$")
```

```
print ("line2")
```

```
line("-",20)
```

```
line("*",20)
```

```
print ("line3")
```

```
line("-",30)
```

```
line("@",30)
```

```
print ("line4")
```

line(nb=5)

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 my_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 1 (ex3)*

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 = 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" (=package 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 `input()` function is the opposite of the `print()`. Its role is to wait for the user to enter something on the keyboard, the string that is passed as parameter being displayed on the screen.

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

Note : on python v2.x, `input()` is replaced by `raw_input()` function

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* **player_number** > **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 be executed)

Taking our previous algorithm, this gives:

```
if ( player_number > 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 », complete the program...

Interactions with a program

Full program:

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

while ( not (player_number == random_number)) :

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

    string = input("Give a number :")
    player_number = int(string)

print ("Good !")
```

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

while (player_number != 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. *if (condition) : ...*
- 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**. *while (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.

Interactions with a program

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 (note : 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 = input("Give a number :")
```

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

```
    return string
```

```
play = "yes"
```

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

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

```
    player_number = number()
```

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

```
        if ( player_number > random_number) :
```

```
            print ("Too big !")
```

```
        else :
```

```
            print ("Too small !")
```

```
        player_number = number()
```

```
play= input("Good ! Do you want to play again (yes/no) ? : ")
```

=> *See tutorial 2*

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
```

Warning, 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>	Put 20 as value of the key 'two' in the sub-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> <i>or list(d.keys()) for python >v3.3</i>	return a list containing all the keys of the dictionary d
<code>d.values()</code> <i>or list(d.value()) for python >v3.3</i>	return a list containing all the keys of the dictionary d
<code>len(d)</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)

The 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 (my_func())
```

```
hello
```

The exceptions

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

Example:

try:

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

except :

```
print ("Oups ! 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>

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 !

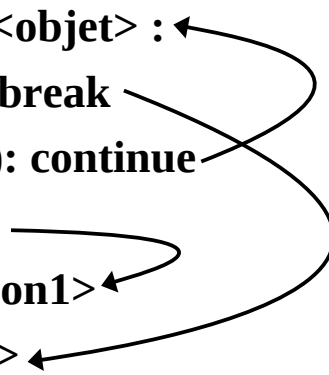
Iterations (=loops)

“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
- "**continue**" 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.

```
for <cible> in <objet> :  
    if (test1): break  
    elif (test2): continue  
    else: pass  
    <instruction1>  
<instructions2>
```



=> go directly outside the loop and execute next instruction

=> go directly to the next iteration of the loop

=> do nothing, go the next line

=> normal execution

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.

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

```
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 3*

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: additional 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 bytes (=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 the 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

Exercise (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 4*

GUI Programming in Python

A graphics library is a software library that specializes in graphics functions. It allows you to add graphical representation to a program.

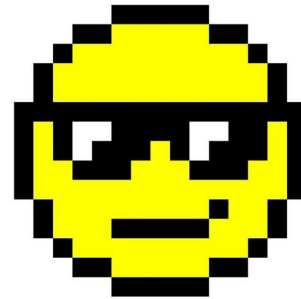
These functions can be classified into three types that have emerged in this chronological order and of increasing complexity:

- 2D element plot libraries
- Users interface libraries
- 3D libraries

GUI Programming in Python

2D element plot libraries

These libraries are also referred to as low-level libraries. They make it possible to draw the basic graphic elements that are lines, polygons and to display pixels which makes it possible to display icons and images.



Graphics libraries can communicate directly with the hardware, i.e. video memory or a graphics card, and use a driver.

The X Window System library under Unix/Linux is typically a library dedicated mainly to this type of functions.

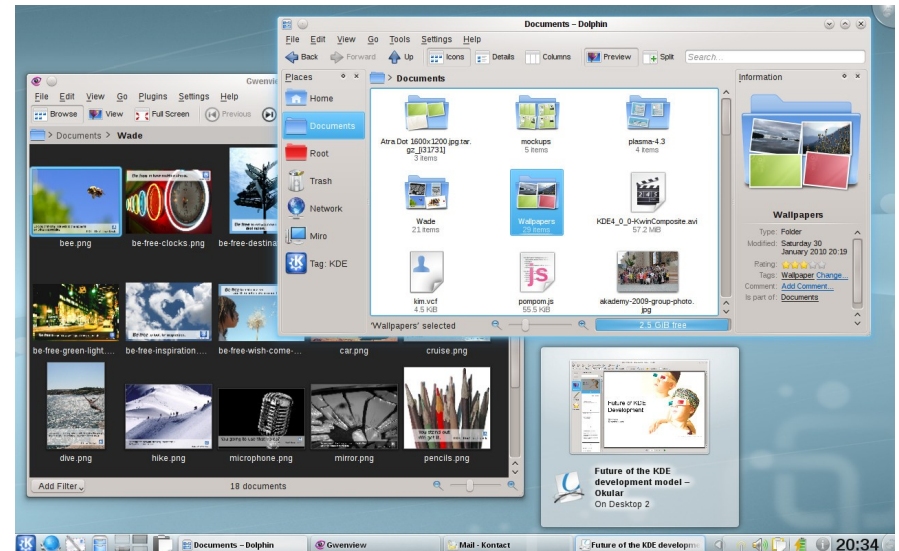
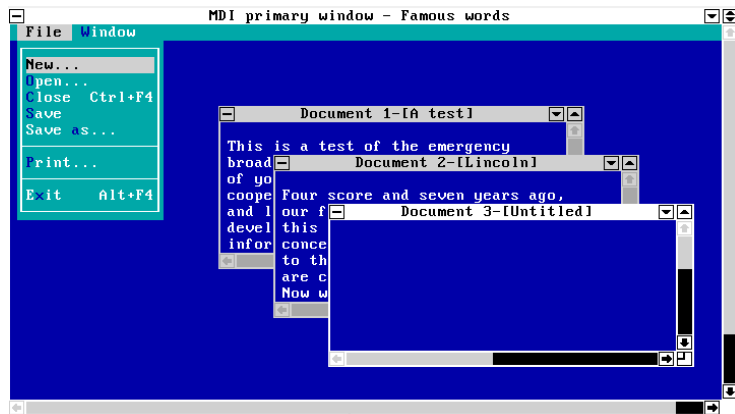
Some older languages such as BASIC included graphical functions as an integral part of the language.

GUI Programming in Python

User interface libraries

User interfaces are the graphical elements that allow the user to interact with the computer. They appeared with the Xerox STAR computer, and are now the basis of computer ergonomics (named for example : Qt, GTK, GNOME, Win32).

They allow through the operating system (Windows, MacOS, Linux, etc.) to provide a graphical representation to read and control the computer's information, using for ie. some windows, buttons, scroll bar, etc.



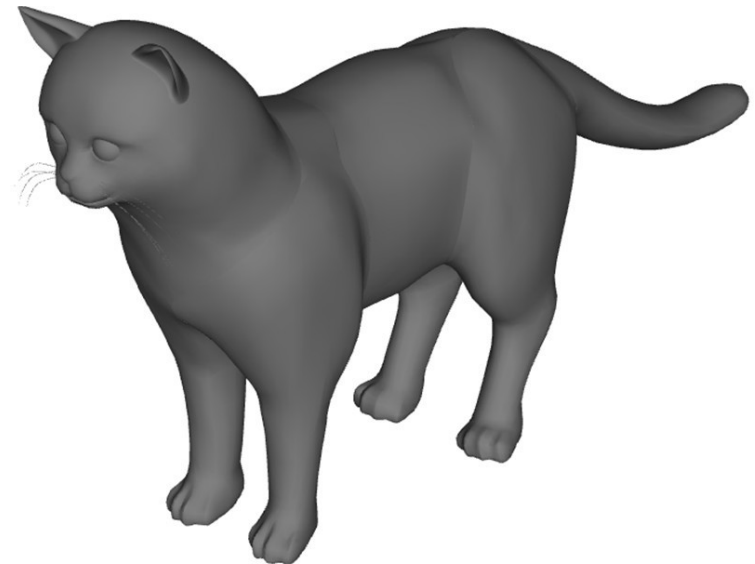
GUI Programming in Python

3D libraries

Last to appear in chronological order, 3D libraries make it possible to synthesize and display 3D images, i.e. to draw elements in volume.

The first 3D library was made by Silicon Graphics: GL became OpenGL later on is one of the best known with Microsoft's DirectX.

Current 3D libraries use an accelerator chip, usually on a dedicated card: the video card



The Tkinter library

Our use of the Python language will be limited to the already vast field of graphical user interfaces (GUI).

- Tkinter is the basic graphical library for the Python language. It is limited but easy to use.
- Tkinter is already included in several Python installation packages, such as ActiveState (see on the web to download actiState)
- Tkinter comes from an adaptation of the Tk graphics library written for Tcl.
- Tk and Tkinter are available on most Unix platforms (including all Linux), Windows and Macintosh.

To use Tkinter, all you need to do (if Tkinter is already installed) is import this library:

```
tkinter import
```

Or (prefer):

```
from tkinter import *
```

Note : for python2.x, use “Tkinter” instead “tkinter” (same library, but slight different name...)

First example - The "Label" widget

from tkinter import * # import all classes (*) contained in the Tk module

fen = Tk() # create a window (instance of the class Tk - "Instance" <=> ~ function in object language)

text = Label(fen, text="Welcome !", fg='red')

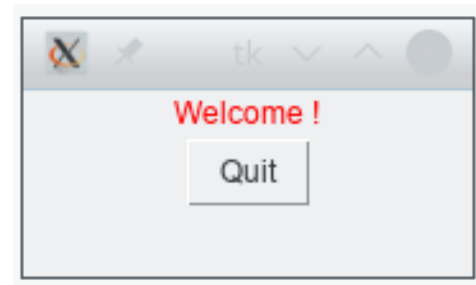
creation of the text object (by instantiation of the widget "Label"). The 1st parameter is the name of the main window (here called "fen"), then the simple text, and finally optional arguments like here the color of the text in red

text.pack() # the method (method <=> function similar in object language) "pack()" displays the element in the window

button= Button(fen, text="Quit", command=fen.quit) # creation of the button object (by instantiation of the Button class), in "fen"

button.pack() # button object placement

fen.mainloop() # makes a loop on the window to prevent it from closing automatically at the end of the program. More precisely "mainloop" manages events (click mouse, keyboard) and the window management system (refresh). This command line must therefore be the last of any program



Presentation of the “Widgets”

Main widgets:

The graphical components of Tkinter are called “widgets”.

There are 15 widget classes in the Tkinter module

Each of these widgets is associated with a large number of options and methods.

We can also link events (=functions) to them

Widget	Description
Button	A classic button, to be used to trigger the execution of any command.
Canvas	A space to arrange various graphic elements. This widget can be used to draw, create graphical editors, and also to implement custom widgets.
Checkbutton	A "checkbox" that can take two distinct states (the box is checked or not). Clicking on this widget causes the state to change.
Entry	An entry field, in which the program user can insert any text from the keyboard.
Frame	A rectangular area in the window where other widgets can be placed. This surface can be colored. It can also be decorated with a border.
Label	Any text (or label) (possibly an image).
Listbox	A list of choices offered to the user, usually presented in a kind of box. You can also configure the Listbox to act as a series of "radio buttons" or checkboxes.
Menu	One menu. It can be a drop-down menu attached to the title bar, or a "pop-up" menu appearing anywhere following a click.
Menubutton	A menu button, to be used to implement drop-down menus.
Message	Allows you to display a text. This widget is a variant of the Label widget, which automatically adapts the displayed text to a certain size or width/height ratio.
Radiobutton	Represents (by a black dot in a small circle) one of the values of a variable that may have several values. Clicking on a "radio button" gives the value corresponding to the variable, and "empty" all the other radio buttons associated with the same variable.
Scale	Allows you to vary the value of a variable in a very visual way, by moving a cursor along a rule.
Scrollbar	"scrollbar" or "scrollbar" that you can use in association with other widgets: Canvas, Entry, Listbox, Text.
Text	Display of formatted text. Also allows the user to edit the displayed text. Images can also be inserted.
Toplevel	A window displayed separately, "above".

Presentation of the “Widgets”

Common Options:

Options	Description
width	Positive value indicating the width of the widget (in pixels or, if font, the size of the font). In null or negative value, then the width is automatic
height	Positive value indicating the height of the widget (in pixels or, if font, the size of the font)
background (bg)	Color of the background of the widget
foreground (fg)	Color of the foreground of the widget
relief	3D effect of the widget. Possibles values are : RAISED, SUNKEN, FLAT, RIDGE, SOLID and GROOVE. This value indicates how the inside of the widget looks like compare to the outside
borderwidth (bd)	Positive value indicating the width of the 3D border to draw inside the widget (if exists, with the option “relief”)
takefocus	Boolean indicating if the widget accepts the focus from the keyboard. <shift – Tab>
highlightcolor	Color of the rectangle that is displayed around the widget when the focus of the keyboard is on it
highlightbackground	Color of the rectangle that is displayed around the widget when the focus of the keyboard is NOT on it
highlightthickness	Positive value indicating the width of the borders rectangle displays by the keyboard focus
font	Font to be use inside a widget
cursor	Mouse cursor to be used for a widget

Presentation of the “Widgets”

Options to read and modify the widgets (3 methods)

get(option)

return the current value of an option as a string

configure(option=value,...), config(option=value, ...)

modify one or more options

keys()

returns a list of all options that can be modified

Manage the position of widgets

Manager	Description
Grid	Display is managed as a table, by organizing the widgets into a 2D grid (into cells, with lines and row coordinates)
Pack	Display a widget relatively to its parents (if existed. Usually link to the right of the parents). Widgets are in “blocks” placed into a frame
Place	Give an absolute position with x and y coordinates (in pixel, into the window)

The widget « Label »

A “Label” widget can display small text, an icon or an image

Note : to display several lines of text, the “Message” widget is more suitable because it offers more formatting options

Options :

Option	Type	Description
text	string	Text displayed in one or more lines. If option “bitmap” or “image” is mentioned, this option is ignored
bitmap	bitmap	“bitmap” to be added into the widget “Label”. If the option ‘image’ is mentioned, this option is ignored. Following bitmaps worlds can be used : error, gray75, gray50, gray25, gray12, hourglass, info, questhead, question, warning, it alos possible to load a bitmap from an XBM file, using for ie “@sample.xbm”
image	image	Image displayed into the widget “Label”. If this option is mentioned, it disabled the options “text” and “bitmap”
justify	constant	Align text lines with : LEFT, RIGHT or CENTER
anchor	constant	Position of text or image into the label. Use : N, NE, E, SE, S, SW, W, NW, or CENTER (default)
wraplength	distance	Define the length to split lines. Value is relative to the main screen display. Default is multi-lines
textvariable	variable	Replace the text in Label with a variable (usually a StringVar). If this variable is modified, the text will also change on the fly
underline	int	Underline the text. Default is no underline

The widget «Button»

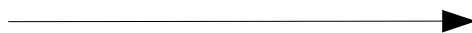
This widget allows the user to do an action when clicking on it. Usually the text or an icon on the button is explicit for this action, ie. dialog box to accept or reject a demand

Methods:

`flash()`: flashes the button between normal and active state.

`invoke()`: simulates a call from the command associated with the button.

Options :



Option	Type	Description
anchor	constant	Position of text or image into the label. Use : N, NE, E, SE, S, SW, W, NW, or CENTER (default). If this option is used, it is good to also use the option padx and pady
bitmap	bitmap	“bitmap” to be added into the widget “Button”. If the option ‘image’ is mentioned, this option is ignored. Following bitmaps worlds can be used : error, gray75, gray50, gray25, gray12, hourglass, info, questhead, question, warning, it alos possible to load a bitmap from an XBM file, using for ie “@sample.xbm”
command	callback	Function or method called when the button is pressed
disableforeground	color	Color of the text or the image when the button is unavailable. The background color is unchanged
image	image	Image displayed into the widget “Image”. If this option is mentioned, it disabled the options “text” and “bitmap”
justify	constant	Align text lines with : LEFT, RIGHT or CENTER
padx, pady	distance	Horizontal margins between the text or the image and the frame of the button
state	constant	State of the button : NORMAL (default), ACTIVE or DISABLED
text	string	Text displayed over the button. If options ‘bitmap” or “image” are mentioned, this option is ignored
textvariable	variable	Replace the text over the button with a variable (usually a StringVar). If this variable is modified, the text will also change on the fly
underline	int	Underline the text. Default is no underline
wraplength	distance	Define the length to split lines. Value is relative to the main screen display. Default is single-line

The widget «Button»

Example: Shows a message after a button click.

```
from tkinter import *
```

```
def info():
```

```
    lab=Label(root, text="I like Python ! ")
```

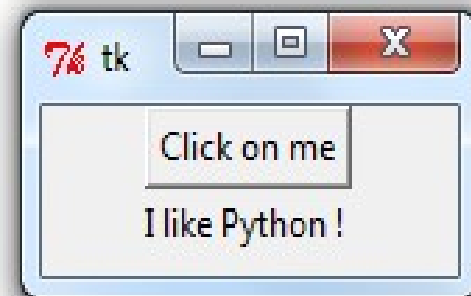
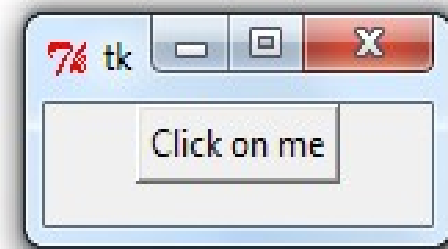
```
    lab.pack()
```

```
root=Tk()
```

```
but=Button(root, text="Click on me", command=info)
```

```
but.pack()
```

```
root.mainloop()
```



Le widget «Text»

The widget “Text” is used to display simple or formatted text documents (various fonts and layouts are available).

This widget allows you to specify the position of the cursor into the text: column line, end of line, INSERT, CURRENT, END, Label, Selection, Mouse coordinates, etc...

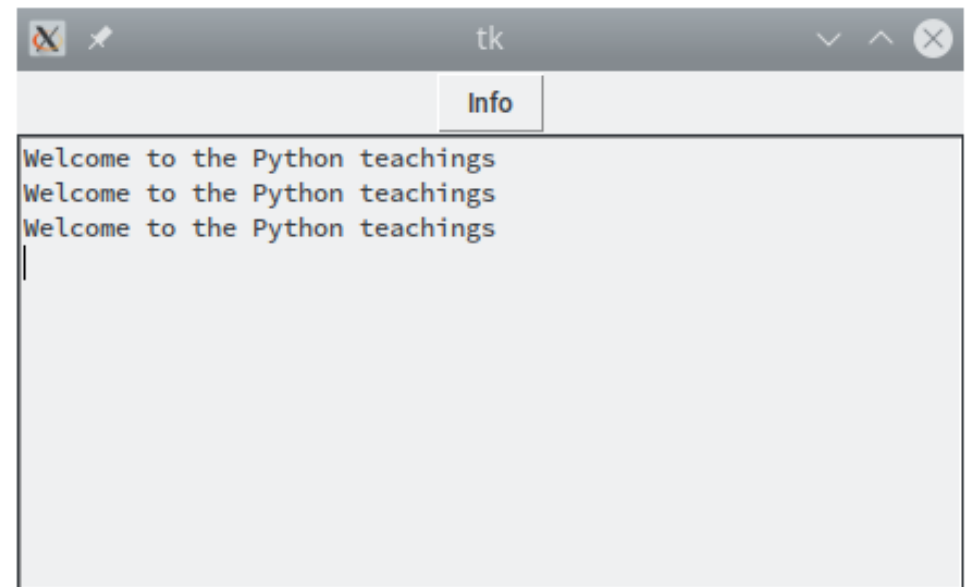
Methods:

- insert (index,text): inserts some text at “index” place (=position of the cursor , ig. number of characters from the begin).
- delete(index), delete(start,stop) : deletes the position character "index" or the whole text between start and stop.
- get(index), get(start,stop): returns the position character "index" or the whole text between start and stop.
- dump(index,options...), dump(start,stop,options...): returns the list of components of the given "index" position widget or the whole text between start and stop.
- index(index): returns the "row/column" index corresponding to the given index.
- ...

=> See the completed list of options and methods on <http://effbot.org/tkinterbook/text.htm>

Le widget «Text»

```
from tkinter import *  
  
def my_info():  
    txt.insert(END, "Welcome to the Python teachings \n")  
  
root=Tk()  
  
but=Button(root, text="Info", command=my_info)  
but.pack()  
  
txt=Text(root)  
txt.pack()  
  
root.mainloop()
```



The widget Text is created and “pack” into the root (=main) windows:

The my_info() function inserts the character string into the text box. It uses the insert() method of the text widget. The END option specifies the insertion point

The widget «Entry»

The “Entry” widget is used to input strings (equivalent to “input()” function in text mode). It gives the user the possibility to enter a single line of text. To enter several lines of text, use the widget “Text”

Methods:

- insert (index,text): inserts text at the specified index level. Use insert(INSERT,text) to insert text at the cursor and insert(END,text) to add at the end of the text
- delete (index), delete (from,to): deletes the character located at the indicated index position or within a given domain. Use delete(0,END) to delete the entire widget text.
- icursor (index): moves the cursor to the position of the specified index. This also makes it possible to modify the INSERT index mentioned above.
- get (): retrieves the content of the input field
- index (index): returns the numerical position of the specified index.

The widget «Entry»

Example:

```
from tkinter import *  
root=Tk()
```

```
my_text=StringVar()
```

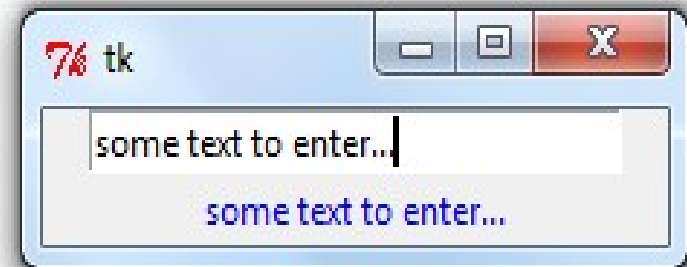
```
lb1= Label(root, textvariable=my_text, width=30, foreground="blue")
```

```
ent=Entry(textvariable=my_text, width=30)
```

```
ent.pack()
```

```
lb1.pack()
```

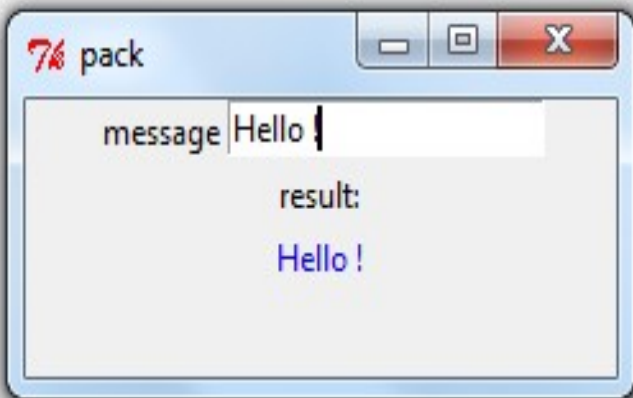
```
root.mainloop()
```



Placing widgets - “pack”

The pack command allows you to place widgets either horizontally (side=left or right) or vertically (side=top).

When you want to switch from horizontal to vertical placement, you use frames.



```
from tkinter import *
```

```
root = Tk()
```

```
root.title("pack")
```

```
f=Frame(root)
```

```
msg="0"
```

```
lb1=Label(f, text="message")
```

```
entre=Entry(f, textvariable=msg)
```

```
lb2=Label(root, text="result:")
```

```
res=Label(root, textvariable=msg, fg="blue")
```

```
# we place the first widgets horizontally in the frame
```

```
lb1.pack(side="left")
```

```
entre.pack()
```

```
# then we place the frame and the other widgets vertically
```

```
f.pack(side="top")
```

```
lb2.pack()
```

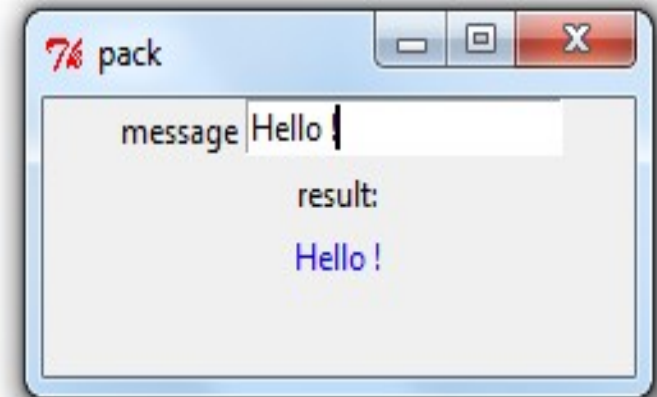
```
res.pack()
```

```
root.mainloop()
```

Placing widgets - “grid”

Same example as previously, but using the command “grid”

```
from tkinter import *  
root = Tk()  
  
msg=""  
lb1=Label(root,text="message")  
entre=Entry(root,textvariable=msg)  
  
lb2=Label(root,text="result:")  
res=Label(root,textvariable=msg,fg="blue")  
  
lb1.grid(row=0,column=0)  
entre.grid(row=0,column=1)  
lb2.grid(column=1)  
res.grid(column=1)  
  
root.mainloop()
```

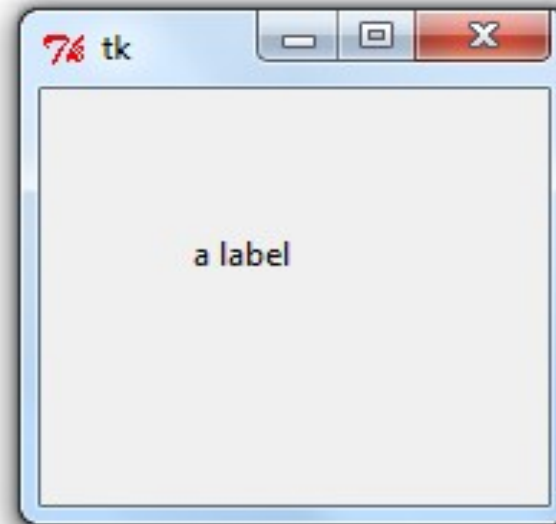


The column and row options allow you to specify the positioning of an object.

Placing the widgets - “place”

“place” allows to place a widget at an absolute position “x” and “y”:

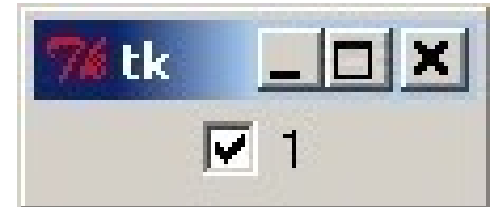
```
from tkinter import *  
root = Tk()  
lb=Label(root,text="a label")  
lb.place(x=50, y=50 )  
  
root.mainloop()
```



The widget «CheckBox»

Example:

```
from tkinter import *  
root=Tk()  
val=1  
chk=Checkbutton(root, textvariable=val, variable=val)  
chk.grid()  
root.mainloop()
```



The widget «RadioButton»

A radiobutton looks a bit like a checkbutton, except that you can associate several radiobuttons to the same variable and define for each of them the value that will be assigned to the variable when it is checked.

Example:

```
from tkinter import *
```

```
root=Tk()
```

```
myvar = 0
```

```
r1=Radiobutton(root, text="1", variable=myvar, value="1")
```

```
r2=Radiobutton(root, text="2", variable=myvar, value="2")
```

```
r3=Radiobutton(root, text="3", variable=myvar, value="3")
```

```
r1.grid()
```

```
r2.grid()
```

```
r3.grid()
```

```
root.mainloop()
```



The widget «Canvas»

A canvas is an area in which you can draw lines, rectangles, circles, arcs, curves or display images.

Example:

```
from tkinter import *  
root = Tk()
```

```
canv=Canvas(root,width=200,height=300)
```

```
canv.create_rectangle((2,2,199,199),fill="white",outline="red")
```

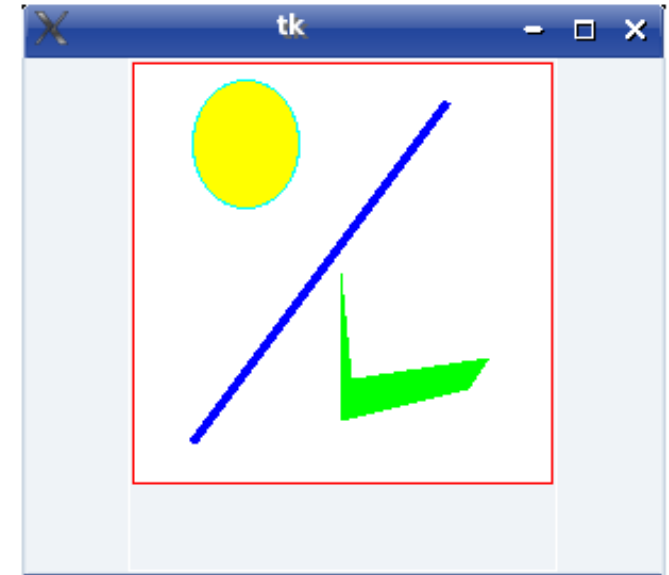
```
canv.create_line((150,20,30,180),fill="blue",width=4)
```

```
canv.create_oval((30,10,80,70),fill="yellow",outline="cyan")
```

```
canv.create_polygon((100,100, 105,150, 170,140, 160,155, 100,170),fill="green")
```

```
canv.pack()
```

```
root.mainloop()
```



Exercise

Rectangles generator:

Make a program that draws rectangles of different sizes based on coordinates entered by a user:

- Make a grey canvas that will serve as a wallpaper (400 by 400 for example)
- Make 4 Entry boxes corresponding to the 4 coordinates to draw the rectangle (upper left and lower right corner: x1, y1, x2, y2)
- Put 4 labels for these 4 boxes (names of the boxes for example)
- Make one button to exit the program and another to draw the rectangle
- This last button will launch a function that will draw the rectangle ("create_rectangle" method to be applied on the background canvas. The coordinates to be passed as arguments are the values of the 4 input boxes (which must be retrieved with the get() method example `x = my_object.get()`)

Note: `fill="one color"` and `outline="another color"` are arguments that can be used for a canvas (rectangle or not)

Correction

```
from tkinter import *
```

```
def trace_rectangle ():                # function to draw rectangles
    x1=ent1.get()                      # get the values from the entries boxes
    y1=ent2.get()
    x2=ent3.get()
    y2=ent4.get()
    canvas.create_rectangle((x1,y1,x2,y2), fill="blue", outline="red")
```

```
fen = Tk()                            # create a window, instance from the classe Tk
```

```
canvas = Canvas(fen, height=400,width=400,bg='dark grey')    # create a canvas as a background frame
canvas.pack(side=TOP)
```

```
# create "entry" boxes in order to put values and labels
```

```
lab1 = Label(fen, text="x1")
lab1.pack(side=LEFT)
ent1 = Entry(fen, width=5)
ent1.pack(side=LEFT)
```

code part 2 ==>

Correction (continued)

```
lab2 = Label(fen, text="y1")  
lab2.pack(side=LEFT)  
ent2 = Entry(fen, width=5)  
ent2.pack(side=LEFT)
```

```
lab3 = Label(fen, text="x2")  
lab3.pack(side=LEFT)  
ent3 = Entry(fen, width=5)  
ent3.pack(side=LEFT)
```

```
lab4 = Label(fen, text="y2")  
lab4.pack(side=LEFT)  
ent4 = Entry(fen, width=5)  
ent4.pack(side=LEFT)
```

```
bouton1= Button(fen, text="Trace my rectangle", command=trace_rectangle) # button to draw  
bouton1.pack()
```

```
bouton2 = Button(fen, text="Quit", command=fen.quit) # button to quit  
bouton2.pack(side=BOTTOM)
```

```
fen.mainloop() # evens manager
```

=> *See tutorial 5*