python™

KEEP
CALM
AND
WELCOME
TO
PYTHON

Who am I?

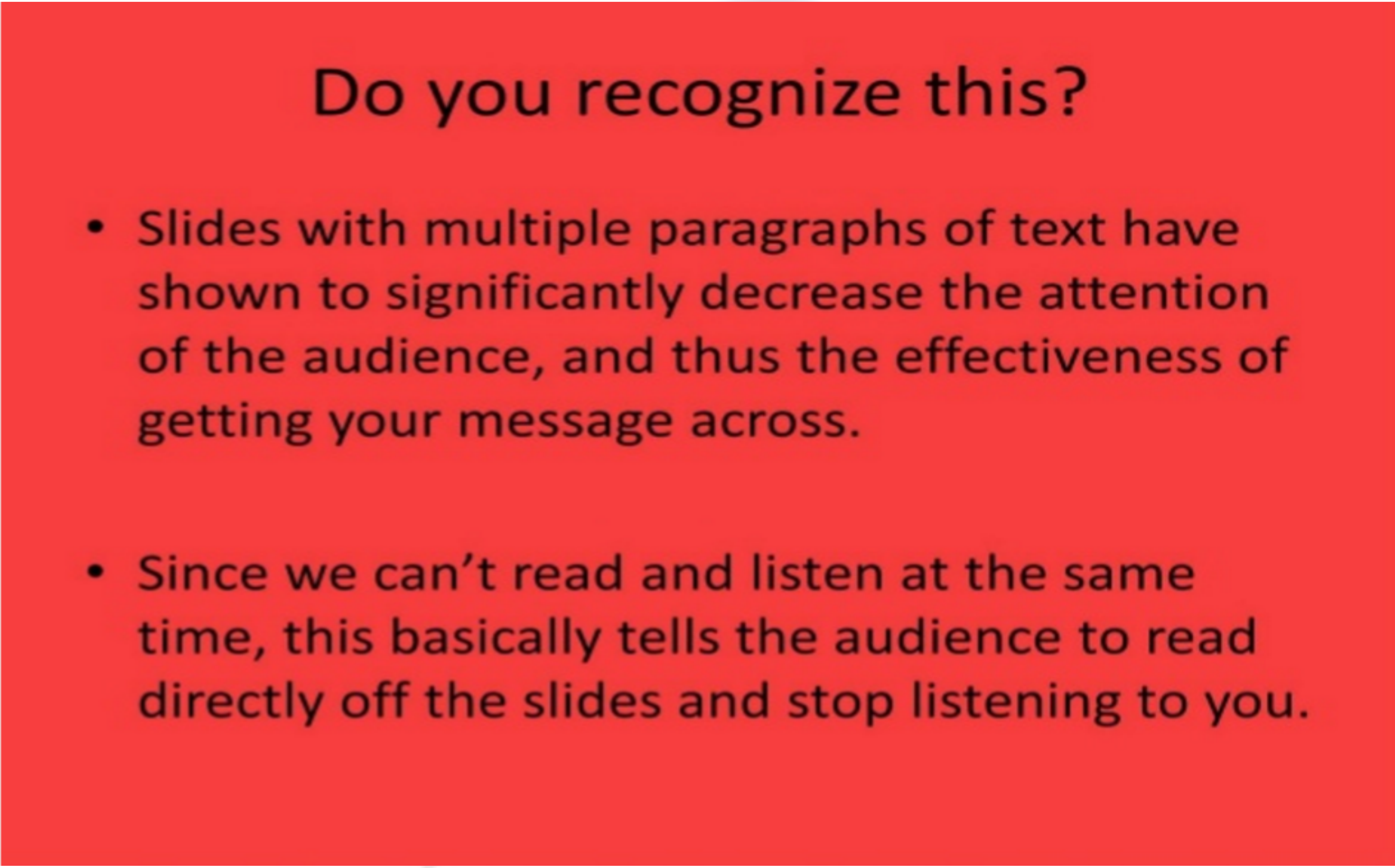
Who are **you**?

- It’s free
- It’s Object Oriented
  - It supports with a easy syntax Polymorphism and Inheritance.
- It’s Portable
  - You can execute the same code on every platform, provided with the Python interpreter.
- It’s simple
- It has a lot of libraries
  - Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. In addition, Python can be extended with both homegrown libraries and a vast collection of

WHY are you here?

  - Database interaction with ODBC
  - Games developing
    - **Pygame, pykyra**
  - Scientific programming
    - **Numpy, Scypy**
- It’s widely used
  - NASA, Yahoo!, Google, Youtube, RedHat,
  - OpenStack it’s written in Python

Images say more than a thousand words



Do you recognize this?

- Slides with multiple paragraphs of text have shown to significantly decrease the attention of the audience, and thus the effectiveness of getting your message across.

- Since we can’t read and listen at the same time, this basically tells the audience to read directly off the slides and stop listening to you.

This is basically the type of slides that you will get here.

A quick introduction

The interpreter

- An **interpreter** it’s a software application that runs other programs. When you write a command in the Python shell, or when you run a Python script, the Python interpreter reads your code and executes the instructions: you can imagine it as a logical layer between the code and the hardware.

- Python writes a intermediate **byte-code**, that is a "native" Python command list, to be executed in the **Python Virtual Machine**.

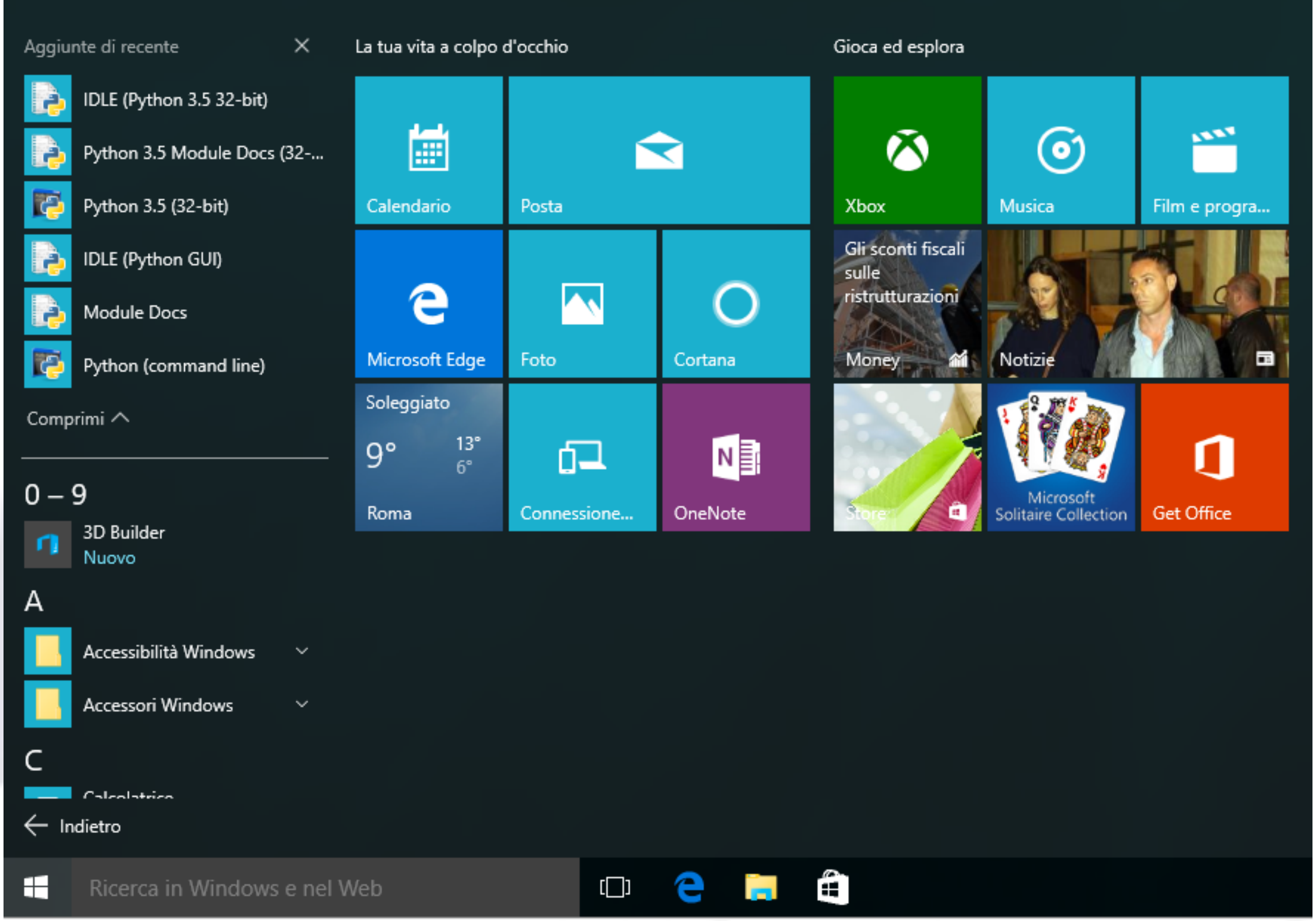
- Python is installed on every Linux distribution and on Apple OSX. If you want a Windows interpreter you can get it from: http://www.python.org

- With Python is quite easy to write code that does not depend from the machine operating system: you can write the code on your Linux server, and than copy it on the Windows desktop. (or viceversa)

Program execution

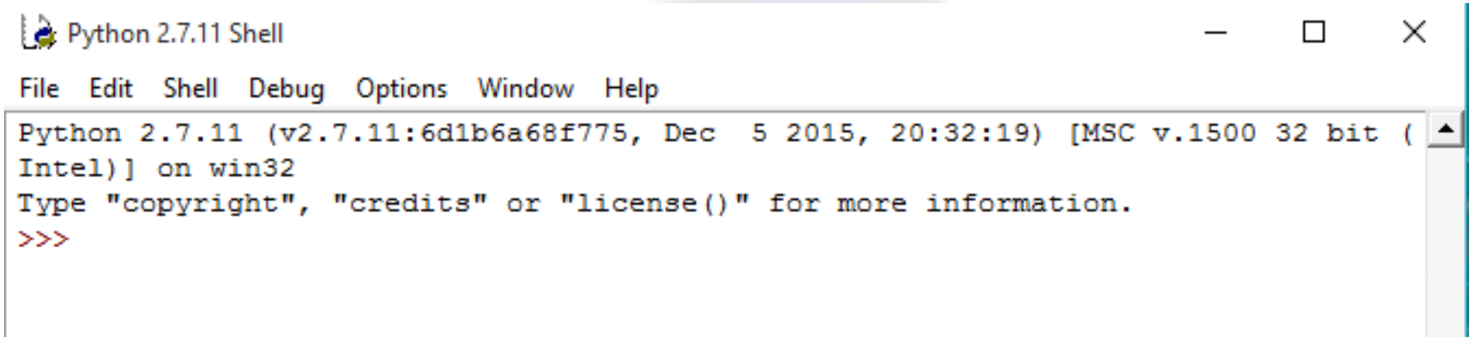
- How does the code execution work in Python? The Python scripts are interpreted in two stages:
  1. A first stage compiles the text code in *byte-code*, a low-level binary format, that is platform independent. The byte code is stored in *.pyc* files: these files will be directly loaded when the script is executed multiple times.
  2. The Python Virtual Machine, an intermediate software layer, runs the byte-code

- There is not a «*make*» phase that translates the text code in binary machine code.
  1. PROS: The code is highly portable, and it is possible to run code from an interactive shell.
  2. CONS: The execution cannot have the same speed that can be reached with a compiled code.



IDLE starts a simple visual editor.
Python CLI starts an interactive session with the interpreter.

What does it mean to write a Python program? All you have to do is write the Python instructions in a text file, preferably with the estension *.py*, and then execute it in the Python interpreter.



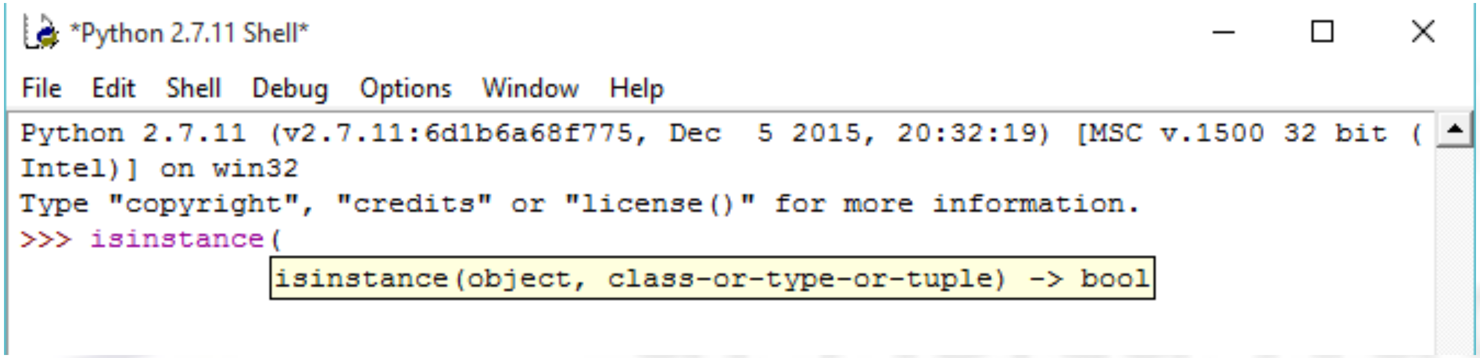
```
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 20:32:19) [MSC v.1500 32 bit (
Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

- With IDLE you can write your code and test it in the Python shell.
- You can input your code after the Python shell prompt, the "triple chevron" **>>>**
- The shell executes the code **immediately**, and shows the results on the screen.
- IDLE "knows" the Python syntax and functions, so it’s able to give you an hint when you use a built-in function, or in general a function that is available in the current namespace.

  For Example: if you write

  `isinstance(`

  you can see that IDLE gives you the function arguments in a tooltip.



```
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 20:32:19) [MSC v.1500 32 bit (
Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> isinstance(
               isinstance(object, class-or-type-or-tuple) -> bool
```

IDLE

- IDLE colours automatically the code text. Default is green for strings, orange for language keywords, purple for built-in functions. You can of course change all the default settings.

- IDLE has an automatic code indentation, and that can be VERY useful in a complex code, given the fact that **in Python the indentation is a part of the language**.

- If you start to type a command, pressing the **TAB** key you can have hints for its completion, as in bash shell. Check with:
  `isin`+**TAB**

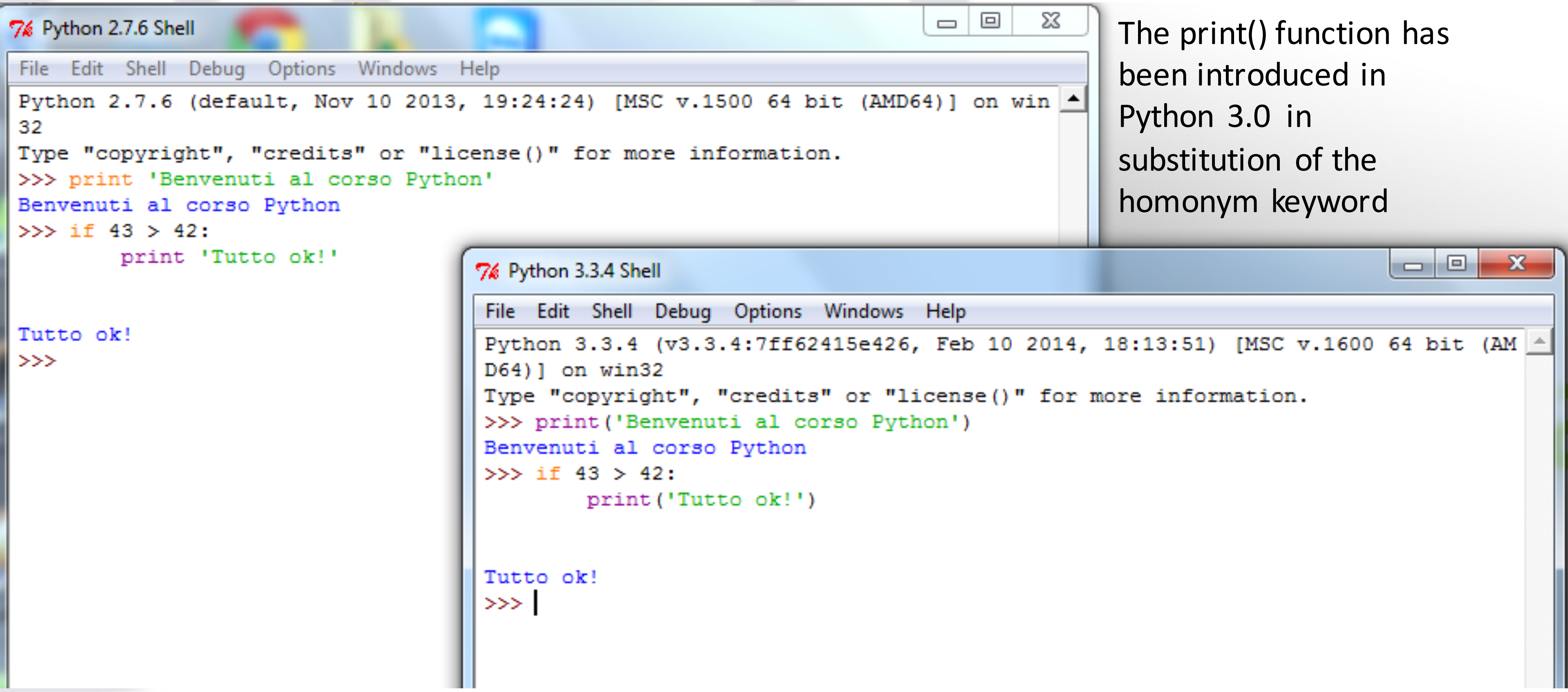
- IDLE mantains a command history: with **Alt-P** (**P**revious) you can recall the code executed, with **Alt-N** (**N**ext) you can browse the history forward. You can also modify the recalled code.

PLEASE note that Python uses CODE INDENTATION to group code that are logical correlated, as functions, if statements, for loops, while loops, and so forth.

Indenting a code line starts a block and unindenting ends it, there are no explicit braces, brackets, or keywords. The indentation level of your statements is significant, and allow you to nest code blocks.

The exact amount of indentation doesn't matter at all, Pythons cares about the relative indentation of nested blocks (relative to each other).

DO NOT to mix tabs and spaces for indentation. If you use tabs only or spaces only, you're fine.



Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help

```
Python 2.7.6 (default, Nov 10 2013, 19:24:24) [MSC v.1500 64 bit (AMD64)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print 'Benvenuti al corso Python'
Benvenuti al corso Python
>>> if 43 > 42:
        print 'Tutto ok!'


Tutto ok!
>>>
```

Python 3.3.4 Shell
File Edit Shell Debug Options Windows Help

```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print('Benvenuti al corso Python')
Benvenuti al corso Python
>>> if 43 > 42:
        print('Tutto ok!')


Tutto ok!
>>> 
```

The print() function has been introduced in Python 3.0 in substitution of the homonym keyword

Scripts

- The IDLE shell can be used also to write Python scripts. A script, or a Python program, is a text file that contains Python instructions to be executed for a specific purpose.

- With *File->New* you can open a new window in which is possible to write the script. The script must be saved before being executed.

- With *File->Open* you can open and modify a already saved script

- If you want to run the script, you can use *Run->Run Module* on the menu of the window that contains it.

Advanced IDLE

- From the *Debug* menu you can enable the built-in debugger

- A debug session is started when
  - the debug is enabled and then
  - the loaded script is executed from *Run->Run module.*

- You can enable a breakpoint with a right click on the code.

- For fast debug it is possible to jump from the error message, with a right click, to the code line that originated it.

- On the Linux distributions a dedicated debugger, `pdb`, is also available.

Let's start with the language

- There are four built-in **numeric types**:

  1. integers `int: 0, 1, -3 (C long int)`

     They have at least 32 bit precision

  2. Long integers `long:0L, 1L, -3L.`

     May have arbitrary length, their limit depends on the amount of memory in the machine.

  3. Double precision real `float: 0., .1, -0.0165, 1.89E+14 (C double)`

  4. Double precision complex `complex: 0j, 1+.5j, -3.14-2j`

- Information on the accuracy and internal representation of floating point for the machine on which you're running a script are available in `sys.float_info`

None

- Python has a Boolean type **bool**, that can take the values `True` or `False`, respectively interchangeable with 1 and 0.

- There is a special variable ***None*** (case sensitive), that identifies a “null object”. It is convenient to use it when you have a variable name but not its value, so the value is **undefined**:

```
answer  = None
#some  updates…
if answer  is  None:
         quit=true
```

- **Pay attention to the test**: to check if a variable is `None` always use

```
if answer is None
if answer is not None
```

The check *if not answer* is dangerous, because it is *True* also if *answer* is an empty string:

with `a = ''; a = []; a= (); a={}; a=0; a=0.0`

```
if a:     #è FALSE
if not a:     #è TRUE
```

- **Mind the operator**: *is* checks the identity of objects, == checks if two objects have the same content

Booleans and Strings

- Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (&&, ||, etc.):

```
t = True
f = False
print type(t)          # Prints "<type 'bool'>"
print t and f          # Logical AND; prints "False"
print t or f           # Logical OR; prints "True"
print not t            # Logical NOT; prints "False"
print t != f           # Logical XOR; prints "True"
```

- Python has great support for strings:

```
hello = 'hello'                            # String can use single quotes
world = "world"                            # or double quotes
print hello                                # Prints "hello"
print len(hello)                           # String length; prints "5"
hw = hello + ' ' + world                   # String concatenation
print hw                                   # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12) # sprintf style string formatting
print hw12                                 # prints "hello world 12""
```

- Python includes several built-in container types: lists, dictionaries, sets, and tuples.

Python lists

Let’s start from *data*

- Almost every serious program works with data!

- Sometimes the data are very simple, and it’s easy to work with them. Sometimes they have a complex structure, and the code that must use and modify them can become quite involved.

- Often it can be useful to accomodate the data in a *list*, that express a **logical** connection between them: a clients list, your friends list, the to-do list, a list of time records, etc..

- For such a common task Python has a dedicate data type: **list**

An example

Let's suppose that our problem data are the following:

I soliti ignoti, 1958, Mario Monicelli, 102 mins

Antonio De Curtis

Vittorio Gassman, Marcello Mastroianni, Renato Salvatori, Claudia Cardinale

I due marescialli, 1961, Sergio Corbucci, 99 mins

Antonio De Curtis

Vittorio De Sica, Gianni Agus, Arturo Bragaglia, Mario Castellani

Uccellacci e uccellini, 1966, Pier Paolo Pasolini, 85 mins

Antonio de Curtis

Ninetto Davoli, Femi Benussi, Gabriele Baldini

At a first glance you can see that we have a complex list, but with an internal structure: each record corresponds to a film, the first line contains some basic informations, the second line contains the name of the protagonist, the third one contains the names of some others actors

`http://bit.do/ListaFilm-txt`

An example

Let’s start from the titles list:

I soliti ignoti
I due Marescialli
Uccellacci e uccellini

and let’s write it in Python

```
>>> film = ["I soliti ignoti", "I due Marescialli", "Uccellacci e
uccellini"]
```

How did we make the translation?

1. Each line has been trasformed in a **string**, using **double quote "**
2. Each list item is separated from the next by a **comma ,**
3. The list items are written between **square brackets []**
4. The list is assigned to an identifier (`film`) with the assignment operator **=**

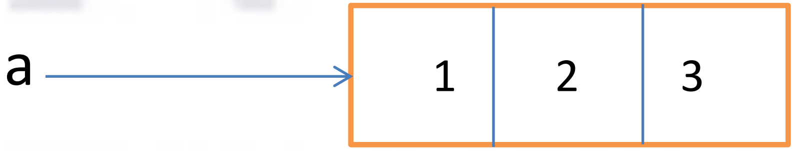
Dynamic Typing

- But… am i not supposed to declare, somewhere before, of WHICH TYPE is the identifier `film`?

- No, because Python is a "**dynamically typed**" language: you can use a new variable without before declare its type.

  In Python a variable identifier is a simple name that refers to an object, whatever is the type of that object.

```
a=[1,2,3]
```



- Be aware that although dynamic, the typing in Python is "**strong**", so you cannot merrily add float and string for example…

Lists items

- When you create a list the interpreter creates a new memory structure, similar to a vector. **The first element has ALWAYS the index 0**, the second 1, the third 2 and so on.

- With these indices, automatically assigned, it is possible to directly address the list items, with the notation:

```
>>> print film[1]
I soliti ignoti
```

- Although this vector similarity, Python lists are something more. A list is a ***collection object***, so it has a bunch of pre-defined functions or methods available.

Example: lists operations

```
>>> classe = ["Anna", 'Giulia', 'Vito', "Michele"]
>>> print classe
['Anna', 'Giulia', 'Vito', 'Michele']
>>> print len(classe)
4
>>> print classe[2]
Vito
>>> classe.append("Rosa")
>>> print classe
['Anna', 'Giulia', 'Vito', 'Michele', 'Rosa']
>>> classe.pop()
'Rosa'
>>> classe.extend(["Rosa",'Gioacchino'])
>>> print classe
['Anna', 'Giulia', 'Vito', 'Michele', 'Rosa', 'Gioacchino']
>>> classe.remove('Vito')
>>> print classe
['Anna', 'Giulia', 'Michele', 'Rosa', 'Gioacch
>>> classe.insert(0, 'Maurizio')
>>> classe.insert(0, 'Vito')
>>> classe.insert(0, 'Elio')
>>> print classe
['Elio,'Vito','Maurizio','Anna', 'Giulia', 'Michele', 'Rosa', 'Gioacchino']
```

len is a Built-In Function(BIF)

append it's a method of list objects that adds an item to the end of the list

pop Removes the item at the given position in the list, and returns it

extend Extends the list by appending all the items in the given list

Remove(x) Removes the first item from the list whose value is *x*.

insert Inserts an item at a given position. The first argument is the index of the element before which to insert

Let’s add other data

I soliti ignoti, 1958, Mario Monicelli, 102 mins
Antonio De Curtis
Vittorio Gassman, Marcello Mastroianni, Renato Salvatori, Claudia Cardinale

I due marescialli, 1961, Sergio Corbucci, 99 mins
Antonio De Curtis
Vittorio De Sica, Gianni Agus, Arturo Bragaglia, Mario Castellani

Uccellacci e uccellini, 1966, Pier Paolo Pasolini, 85 mins
Antonio de Curtis
Ninetto Davoli, Femi Benussi, Gabriele Baldini

This is our initial list:

```
>>> film = ["I soliti ignoti", "I due Marescialli", "Uccellacci e
uccellini"]
```

Now we want to add the year of production, using the methods of the previous slide.

But…

- Is it possible to add numeric items in a list that already contains string items?
- **YES**: a Python list can contain objects of different types. In fact you can do more than mix numbers and strings: you can store in a list data of **any type**.
- The list has to be thought of as a set of logically related objects : Python obviously can not know by what logic they are related, and provides just the tool to group them together, regardless of type.
- Think about the list as a high-level "collection": the data type of items is not important for the list. All that Python needs to know is that you have created a list, you gave it a name, and inside there are data.

Using the insert and append commands , how can I trasform the list:

```
>>> film = ["I soliti ignoti", "I due Marescialli", "Uccellacci e
uccellini"]
```

In the following list?

```
["I soliti ignoti", 1958, "I due Marescialli", 1961, "Uccellacci e
uccellini", 1966]
```

And how am I supposed to create it from scratch , as it is in this second version ?

```
>>> film.insert(1,1958)
```

Remember that the first parameter is the index of the item BEFORE of which the insertion occurs.

```
>>> film.insert(3,1961)
```

After the initial insertion the list expands, so for the second insertion the new element must be taken into account .

```
>>> film.append(1966)
```

```
>>> film = ["I soliti ignoti", 1958, "I due Marescialli", 1961, "Uccellacci
e uccellini", 1966]
```

Loops and conditions

Working with data

- It is often necessary to iterate through the list items, in order to be able to perform some operation on them.

- To do this you can use a `for` loop . This loop is designed in Python to work on lists OR on any other object that Python considers **iterable**.

- **An iterable object is a container capable of returning its items one by one.**

  Objects of this type are all objects that have one of the following functions defined:
  `__iter()__`
  `__getitem()__`

  When an iterable is passed to the BIF `iter()`, which happens automatically inside a for loop, it returns an ***iterator***.

- An *iterator* It is an object that represents a data stream, and enables the programmer to traverse the container.

- In other words, in Python, an iterable is an object which can be converted to an iterator, which is then iterated through during the for loop; **this is done implicitly**.

For loop

```
for    <target identifier> in <iterable>:
       <target processing code>
```

The keyword **for** marks the beginning of the cycle.

The `<target identifier>` it's the variable name that will contain the iterable elements.

The keyword **in** separates the target name from the iterable

A colon **:** indicate the start of the code that processes data

The code **MUST be indented** below the for line.

The target identifier is not previously declared , and it is similar to any other name in Python; when the cycle goes on, the list gives its items to the for loop, and each item in the list is assigned to the target, one after the other.

Numeric for loop

```
for   <target identifier> in range(n):
      <processing code>
```

The **range(n)** BIF allows to realize a cycle that is repetead a defined number of times; it returns an iterable that contains the integers ranging **from 0 to n-1**

```
>>> for num in range(4):
        print num
```

In Python 2.7 while range creates a complete iterable, so if you do `range(1, 10000000)` it creates a list in memory with 9999999 elements, the function **xrange(n)** is a sequence object that evaluates lazily.

List loops

A loop over all the elements of a list:

```
>>> film = ["I soliti ignoti", 1958, "I due Marescialli", 1961, "Uccellacci
e uccellini", 1966]
>>> for item in film:
        print 'item is ', item
```

If you must know the **location** of the item within the list:

```
>>> film = ["I soliti ignoti", 1958, "I due Marescialli", 1961, "Uccellacci
e uccellini", 1966]
>>> for index in range(len(film)):
        print 'film[%d]=%s' % (index, film[index])
```

You can iterate over multiple lists or tuples simultaneously using the **zip** function:

```
>>> for x, y, z in zip(xlist, ylist, zlist)
        #work with x,y,z
```

While loop

What is done with the code:

```
>>> film = ["I soliti ignoti", 1958, "I due Marescialli", 1961, "Uccellacci
e uccellini", 1966]
>>> for each_item in film:
        print each_item
```

Can also be done with a **while** loop:

```
>>> count = 0
>>> while count < len(film):
        print film[count]
        count = count + 1
```

If you are using a while loop **you** have to worry about the "status information" , and so you need to use a variable that allows to end the cycle when the list is exhausted. With the for loop this it is not necessary because everything is handled by the Python interpreter.

while/else - for/else

- A *while* loop can have an additional statement ***else***:

```
i=0
while i<len(L):
        print 'at index', i
        i=i+1
else:
        print X, ' not found'
```

The else code block is executed when the condition of the while loop becomes `False`, also for the first time, then **unless there is a break in the loop it is always executed**.

- A for statement can also have an else block that, unless the cycle contains a break instruction, it is always executed at the end of the loop.

Lists of lists

- We have already seen that the lists can contain data of any type, and also of mixed type: a logical extension of this statement is that they can also contain lists, and indeed it is true!

  Let's return to our example data:

  I soliti ignoti, 1958, Mario Monicelli, 102 mins
  Antonio De Curtis
  Vittorio Gassman, Marcello Mastroianni, Renato Salvatori, Claudia Cardinale

- We can view this as a list of informations on a film, which contains the list of actors, which contains the list of secondary actors.

```
>>> film = ["I soliti ignoti", 1958, "Mario Monicelli", 102, ["Antonio De
Curtis",
["Vittorio Gassman","Marcello Mastroianni","Renato Salvatori","Claudia
Cardinale"]]]
```

Lists of lists

- Nested lists can be manipulated with their list methods, and you can access their data with the square bracket notation already seen. What will be the result of the print instruction?

```
                   0            1           2            3
>>> film = ["I soliti ignoti", 1958, "Mario Monicelli", 102,
         0                      0     4           1  1
["Antonio De Curtis", ["Vittorio Gassman","Marcello Mastroianni",
        2                  3
"Renato Salvatori","Claudia Cardinale"]]]
>>> print film[4][1][3]
Claudia Cardinale
```

- What happens if we use a `for` loop to iterate over the items of this list containing other lists? To check that you can insert this list in an IDLE session and try to print each element with the for loop that we saw before (`for each item in list`)

Lists of lists

```
>>> film = ["I soliti ignoti", 1958, "Mario Monicelli", 102, ["Antonio De
Curtis",["Vittorio Gassman","Marcello Mastroianni","Renato
Salvatori","Claudia Cardinale"]]]
>>> print film
['I soliti ignoti', 1958, 'Mario Monicelli', 102, ['Antonio De Curtis',
['Vittorio Gassman', 'Marcello Mastroianni', 'Renato Salvatori', 'Claudia
Cardinale']]]
>>> for each_item in film:
        print each_item
I soliti ignoti
1958
Mario Monicelli
102
['Antonio De Curtis', ['Vittorio Gassman', 'Marcello Mastroianni', 'Renato
Salvatori', 'Claudia Cardinale']]
```

The for loop prints only the elements of the "primary" list: when it finds the nested list it does not see any difference (is only another object after all) and it prints the list. If you want to do something different, as print the items of the nested list, you need to **check** whether the item is a list and act accordingly.

if...elif...else

```
if   <some condition>:
     <TRUE code>
else:
     <FALSE code>
```

The keyword `if` marks the beginning of the decision-making process

A colon `:` follows the condition that must be met

The keyword `elif <other condition>:` marks the beginning of another condition check

The keyword `else:` marks the beginning of the alternative

The code that processes data must be indented under the lines of if and else

The test *if var* returns *false* if *var* is

1. None
2. The number 0
3. The boolean False
4. An empty string ('')
5. An empty list ([])
6. An ampty tupla (())
7. An empty dictionary ({})

Comparisons

- There is a source of potential confusion when you do a comparison and do not pay attention to not mix strings and numbers.

```
>>> b='1.2'
>>> if b<100:
        print b, '<100'
    else:
        print b, '>=100'
1.2 >= 100
```

- You can have the previous situation if, for example, you load `sys.argv[1]` in a variable `b` and pass 1.2 as the first argument of the command-line script. Here the test b<100 is always FALSE. `b` is a string, and you are comparing it with an integer. Python does not give any error message. The correct syntax is:

```
if float(b)  < 100:

if b< str(100):
```

- Pay attention also to the integer division! As in many other languages the division between two integers gives an integer, which is often not what you want ...

Type check

But how can you check the type of the object pointed by a given name? You can use the BIF `isinstance()`.

```
>>> names = ["Domenico", "Angela"]
>>> isinstance(names, list)
True
>>> num_names = len(names)
>>> isinstance(num_names, list)
False
```

If you want to know all the BIF, you can use:

```
>>> dir(__builtins__)
```

The **`dir`** command can be used to have a list of all the "strings" attached to a particular object.
`__builtins__` it's the name of a ***module*** that contains all the BIFs. **The same command can be used on any object to get information on its properties**. To find out more details about a single function (or any object), you can use the **`help`** command, or invoke an interactive session with **`help()`**

```
>>> help(isinstance)
```

Application

This is our standard code that print every list item, without type checking

```
>>> film = ["I soliti ignoti", 1958, "Mario Monicelli", 102, ["Antonio De
Curtis",["Vittorio Gassman","Marcello Mastroianni","Renato Salvatori",
"Claudia Cardinale"]]]
>>> for each_item in film:
        print each_item
```

Now we can modify it to print the nested lists items.

```
>>> for each_item in film:
        if isinstance(each_item, list):
            for nested_item in each_item:
                print nested_item
        else:
            print each_item
```

Processes the initial list

If it is a list, print its content with a new target

This could be a first approximation.

What is its fault?

Correction

In the example there are three levels of nested lists. You must then re-modify the code to include the third level.

```
>>> for each_item in film:
        if isinstance(each_item, list):
            for nested_item in each_item:
                print nested_item
                if isinstance(nested_item, list):
                    for deeper_item in nested_item:
                        print deeper_item
                else:
                    print nested_item
        else:
            print each_item
```

But ... what if we want to modify the data and add another nested level?
We should modify the code again, and make it increasingly complicated to manage ...

Now it’s time to introduce the **functions**

Functions, variables, assignments

Functions

```
>>> for each_item in film:
        if isinstance(each_item, list):
            for nested_item in each_item:
                if isinstance(nested_item, list):
                    for deeper_item in nested_item:
                        print deeper_item
                else:
                    print nested_item
        else:
            print each_item
```

Same “for loop”

In the print loop a lot of code is repeated. In these cases it is advisable to try to identify the common behavior, and include it in one reusable function, to improve the readability of the code and its maintenance.

Functions

In Python, a **function** is a normal block of code, which can also have input arguments.

You define a function using the keyword **`def`**, providing a name for the function and specifying a list, even empty, of input arguments in parentheses

```
def   <function name>(<argument(s)):
         <function code>
```

The parentheses are mandatory, unlike the arguments. The code of the function must be indented as usual

It does not exist in Python, unlike what happens in C ++, the concept of "function overloading": the number, name or type of arguments can not be used to distinguish between two functions with the same name, **then the name uniquely identifies the function**.

Usually the functions are grouped into **modules**.

Back to the example

The function that we need must take a list as input, and process every item. If the item is not a list, it must print it. If the item is a list, the function must call another copy of itself, with the examined item as argument.

So what we need is a **recursive function**, namely a function that calls itself within its own code.

Say that the function is called **print_lol ()**, and it takes one argument, the list that it must print on the screen. How could you complete the following code?

```
>>> def print_lol(the_list):
         for each_item in the_list:
             if  isinstance(each_item, list):
                  print_lol(each_item)
             else :
                  print each_item
```

As you have noticed the code size has been reduced, the comprehensibility is increased and therefore the maintenance of the code has become much simpler.

And now, even though in the data appears a new level of nested list, there is no need to change the processing code.

Python variables

- In Python when you assign a variable you are making sure that a REFERENCE to an OBJECT has a NAME which identifies it. An assignment then creates a reference, or a pointer to a memory area, and a name.

- A variable name does not has an intrinsic type assigned (as in C `int i`). **The type is in the OBJECTS**, and Python automatically determines the type of the reference (the variable) based on the type of data referenced.

- A reference is created the first time that appears to the left of an assignment expression:

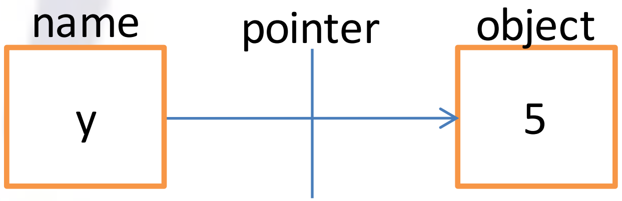
```
>>> x=3
```

- The reference is then deleted by the **garbage collector** after each name attached to it results unused, and an object is deleted if there are no more references pointing to it.

- **The variables must be assigned before they are used in the code.**

What happens when you create a variable?

```
>>>  y=5
```



name

pointer

object

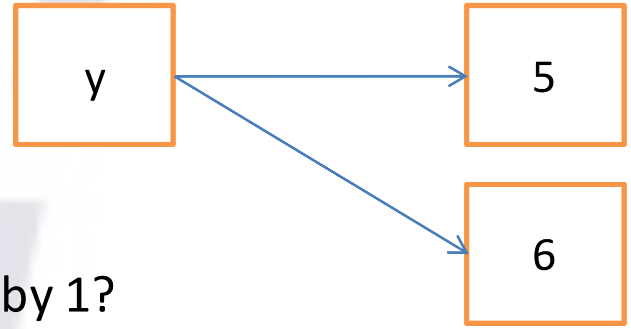
y

5

1. First of all, an integer 5 it's created and placed in memory
2. The name y is created
3. The name y is assigned to a reference to the memory location that contains the new number 5
4. Now when we say that "*the value of y is 5*" we mean that the name y is assigned to a reference that points towards a memory location, which contains the object 5
5. The object 5 that we have created is an integer. Python data types **int, float, and string** are called "**immutable**". This does NOT mean of course that we cannot change y, we can safely change the object towards it points:

Assignment

```
>>> y += 1
>>> print y
6
```



What happens when the value of y is increased by 1?
Python…

1. …searchs the pointer whose name is y.
2. …retrieves the value (5) of the object to which the reference points
3. …computes the sum 5+1, generates a **new** integer object 6 in a **new** memory location
4. …assigns the name y to a reference towards the new memory location
5. …deletes the old memory location (5), **if there are no pointers left pointing at it**.

Immutable assignment between variables

```
>>> x=3          #creates 3, x points to 3
>>> y=x          #creates the name y, that points to 3
>>> x=4          #creates 4, x now points to 4
>>> print y
3                #y points yet to 3
```

- When you modify the data x, you change the reference, but you do NOT touch the reference named y. Everything is OK…

- But in Python there are other data types that are **mutable**, or changeable: lists, dictionaries, and user defined class data types

Mutable assignment between variables

- The mutable data is modified directly without changing the memory location in which they are housed.

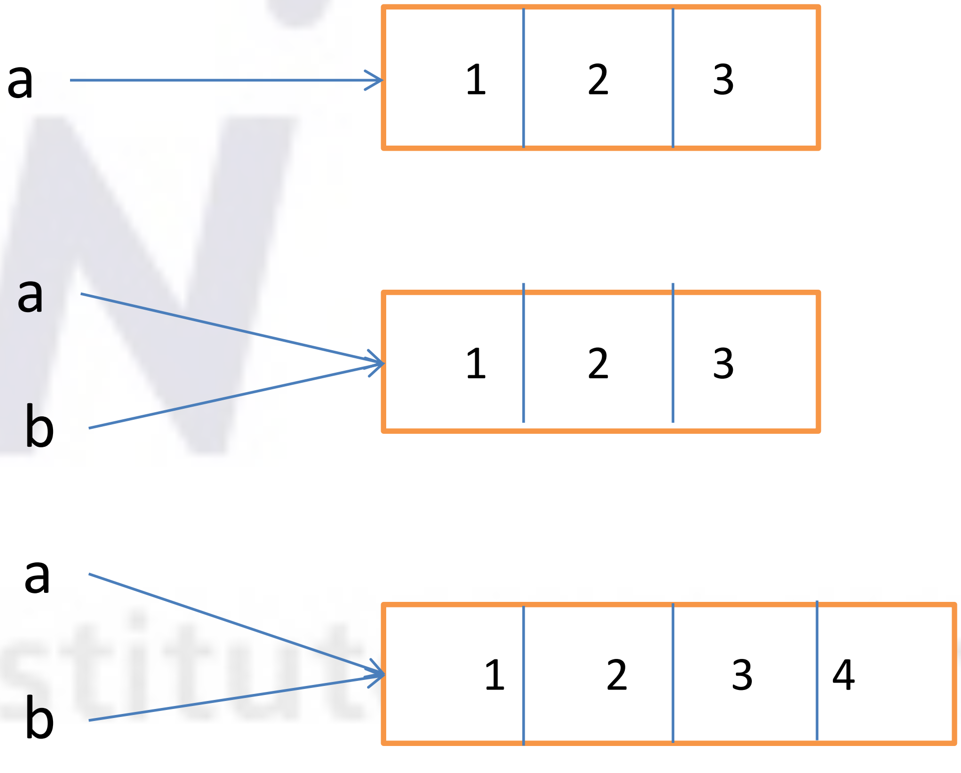
```
>>> a=[1,2,3]          #a points to the list[1,2,3]
>>> b=a                #b point to the same list object
>>> a.append(4)        #this modifies the list object
>>> print b
[1,2,3,4]              #b is also changed
```

- To see if two names point to the same object, you can use the **is** operator, which compares objects; You can also use the **id** operator to have the explicit object's unique numeric identifier.

- This behaviour strongly influences the use of functions input variables.

Intuition

```
a=[1,2,3]

b=a

a.append(4)
```



Functions and variables

	Call by Value	**Call by Reference**
Copy	Duplicate Copy of Original Parameter is Passed	Actual Copy of Original Parameter is Passed
Modification	No effect on Original Parameter after modifying parameter in function	Original Parameter gets affected if value of parameter changed inside function

C and C ++ use by default the "call by value" schema, by making a copy of the variables within the function, but you can use pointers and get the "call by reference".
This last behaviour is standard in Fortran.
So what happens in Python?

Functions and input variables

- Python functions argument passing is a sort of «**call by assignment**», in the sense that an assignment operator is applied between the argument and the variable used in the call.

- **Concretely:** the input function parameters are passed with a «*call by reference*» (pointer), if the data type is mutable, with a «*call by value*» (copy) if the data type is immutable.

- It is then a good practice to always return every parameter modified inside the function. Note that with a single return statement you can return any number of variables.

Example

```
>>> def change1(some_list):
        some_list[1]  = 4

>>> x = [1,2,3]
>>> change1(x)
>>> print x
```

What is the value of x?   `x = [1,4,3]`

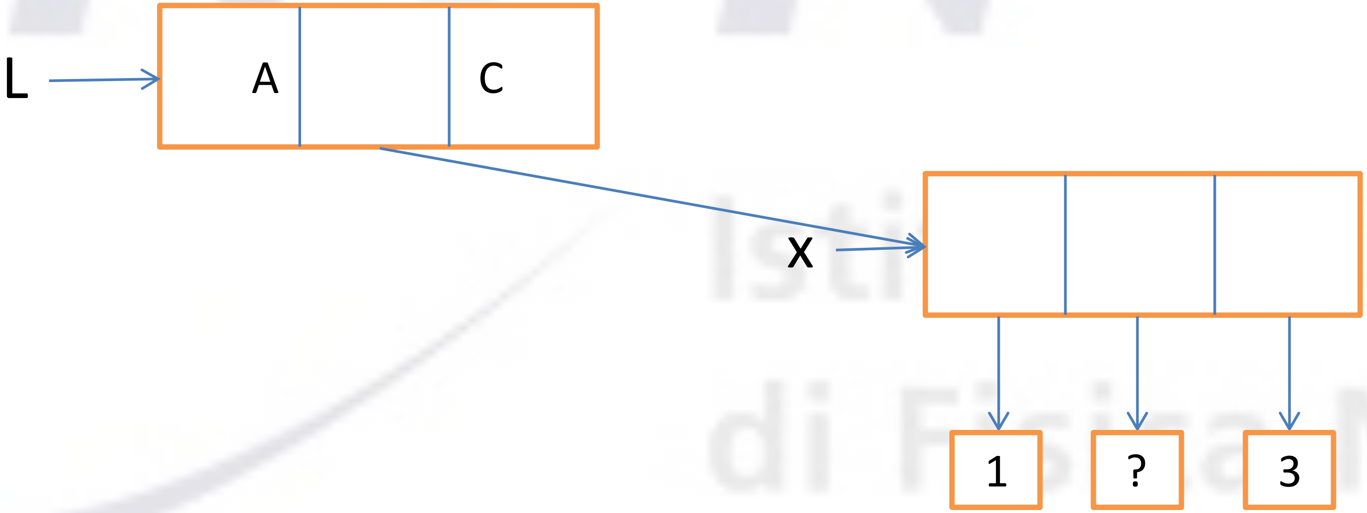
```
>>> def change2(x):
       x = 0

>>> y = 1
>>> change2(y)
>>> print y
```

What is the value of y?   `y = 1`

Shared references

```
>>> x = [1,2,3]
>>> L = ['A',x,'C']
>>> L
['A', [1,2,3], 'C']
>>> x[1] = 'sorpresa!'
>>> L
['A', [1,'sorpresa!',2], 'C']
```

The change in the object pointed by x is visible from every reference that point to the same object.



Copy

But then how is it possible to make a **simple copy**?

```
>>> list2=list1
```

The two names have the same reference to the same object. A change in one of the lists influences also the second list

```
>>> list2=list1[:]
```

Here list1 and list 2 are two **independent copies**, two references to two different objects

```
>>> list2=[]
```

Create an empty list, useful if you want to fill it inside a processing code.

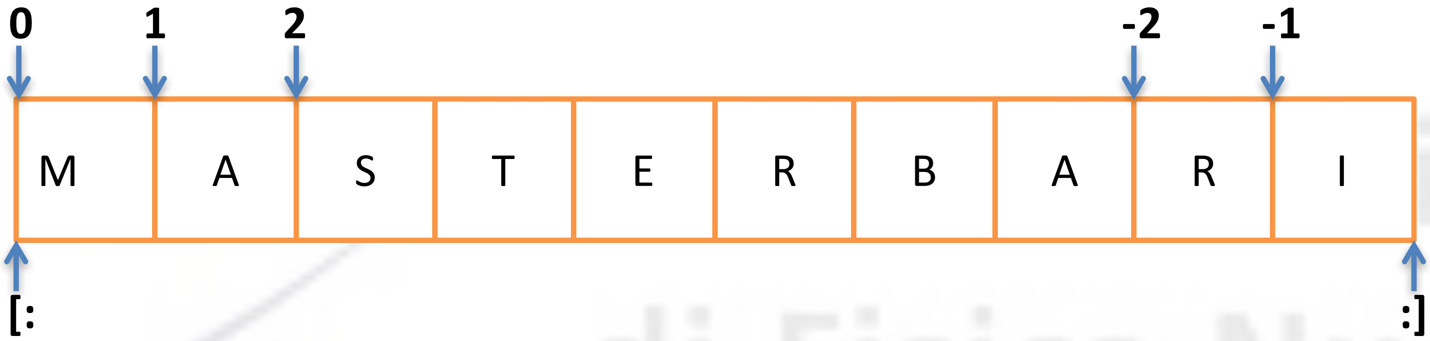
List slicing

1. The **starting** index is always **included**, the *final* index is always *excluded.*

2. If you assign a value to a slice you change the original list **on site**.

3. If you assign a slice to a variable, you create **a copy** of the original list, with the indices specified in the slice.

4. Always remember that the first index is ZERO

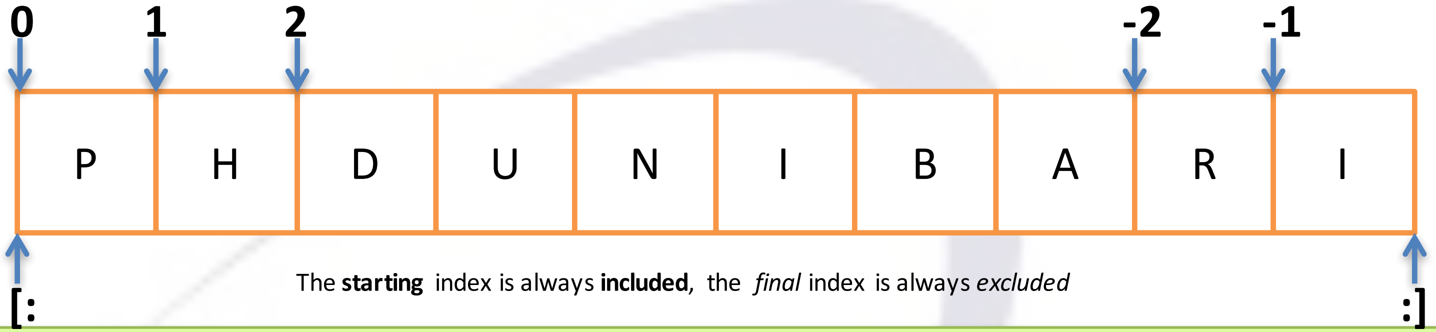


0 **1** **2** **-2** **-1**

M A S T E R B A R I

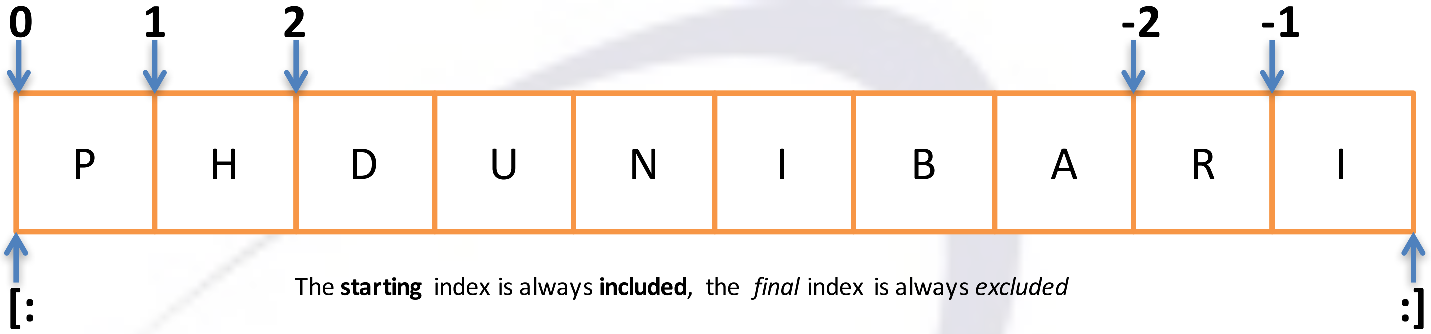
[: **:]**

[start:end]
The indices indicate where the knife "cuts"



The **starting** index is always **included**, the *final* index is always *excluded*

```
>>> a = 'demonstrate slicing in Python'.split()
>>> print a
['demonstrate', 'slicing', 'in', 'Python']

>>> a[-1]     # the last entry
'Python’

>>> a[:-1]    # everything up to but, not including, the last entry
['demonstrate', 'slicing', 'in']    #Equal to a[0:len(a)-1]

>>> a[:]      # everything
['demonstrate', 'slicing', 'in', 'Python']

>>> a[2:]     # everything from index 2 and upwards
['in', 'Python'] #Equal to a[2:len(a)]

>>> a[-1:]    # the last entry
['Python']

>>> a[-2:]    # the last two entries
['in', 'Python']
```



0 1 2 -2 -1

P H D U N I B A R I

[: The **starting** index is always **included**, the *final* index is always *excluded* :]

```
>>> print a
['demonstrate', 'slicing', 'in', 'Python']
>>> a[1:3]                     # from index 1 to 3-1=2
['slicing', 'in']
>>> a[:0] = 'here we'.split()  # add list in the beginning (to the
                               # first, not included)
>>> a
['here', 'we', 'demonstrate', 'slicing', 'in', 'Python']
>>> b = [2.0]*6                # create list of 6 entries,each equal 2.0
>>> b
[2.0, 2.0, 2.0, 2.0, 2.0, 2.0]
>>> b[1] = 10                  # b[1] becomes the integer 10
>>> c = b[:3]
>>> c
[2.0, 10, 2.0]
>>> c[1] = 20                  # is b[1] affected?
>>> b
[2.0, 10, 2.0, 2.0, 2.0, 2.0]  # no c is a copy of b[:3] (slicing is a copy)
>>> b[:3] = [-1]               # first three entries replaced by one entry
>>> b
[-1, 2.0, 2.0, 2.0]
```

tuples

- With the round brackets you can define a special type of lists, **tuples**: like lists they are iterables, but with the particularity that they are **immutable**.

- Another particular example of immutable list is the **string**.

- **You can not change any element of an immutable sequence (tuple or string) without destroying and recreating it, then changing its memory location.**

- A tupla can act as a constant list in the code.

- If the tuple contains a list, can you change the values of the list on-site?

```
>>> mylist=[1,2,3]
>>> mytupla=(4,mylist,5)
>>> mylist[1]='check’
>>> mytupla
(4, [1, 'check', 3], 5)
>>> mytupla[0]=8
TypeError: 'tuple' object does not support item assignment
```

Variables assignation from tuples or lists

- You can extract the items of lists and tuples in separate variables, with the following syntax (here `arglist` is a list or tuple of **three** items):

```
>>> [filename, plottitle, psfile] = arglist
>>> (filename, plottitle, psfile) = arglist
>>> filename, plottitle, psfile = arglist
```

- If on the left there is ONE variable, then we have a list assignment, if there are three variables we assign the list items to the variables, but if there are two, or four or more variables we have an error.

- The **`in`** operator can be used to verify if a list contains a specific item :

```
>>> list=['a','b','c']
>>> var='c'
>>> if var in list:
        print 'Eureka!'
```

It can be used also in the for loops.

Code execution model

- Python executes the code from the beginning of the module or script linearly towards the end.
- References are created when they are processed: *wrong code that it’s not processed does not give any error* ("If it's not tested, it's broken.")

```
>>> x = 3
>>> if x > 5:
        show config()

>>> show_version()
>>> def show_version():
        print 'Version 1.0a'

>>> def test():
        show_version()
>>> def show_version():
        print 'Version 1.0a'
>>> test()
```

The function is NEVER called: we do not have error

This code gives error: you cannot **use** the function `show_version()` before its definition.

This code is ok: when `test()` is executed the `show_version()` function is alredy defined

and/or

1. **`X or Y`**
   If X is `False`, returns Y (that can be `True` or `False`), else returns X

2. **`X and Y`**
   If X is `False`, returns X, else Y

3. **`not X`**
   If X is `False`, returns `True`, else `False`

- The and and or operators are "**short circuits**", in the sense that
  - `or` evaluates the second argument **only if** the first is `False`
  - `and` evaluates the second argument **only if** the first is `True`

- `not` has lower priority than non-boolean operators, so

  ```
  not a==b
  ```

  is interpreted as

  ```
  not (a==b)
  ```

Short circuit

- The operator
  `x and y` (*short circuit and*)
  is equivalent to
  `if x then y else False`

- The operator
  `x or y` (*short circuit or*)
  is equivalent to
  `if x then True else y`

- There are two essential practical applications, think for example to the AND:
  1. The first expression checks the need for intensive computation, which is performed in the second expression
  2. The first expression ensures the occurrence of a condition necessary to the calculation of the second espression
- Ther might also be some problems:
  – If the second expression is a function that must perform operations that are expected independently from the first expression