# Introduction to Python - Part II
# CNV Lab

Paolo Besana and James Bednar

29 January - 2 February 2007

- ▷ *Open a terminal window, and go to the folder "`MyPython`" that you created during the first tutorial (or make a new one for this tutorial).*
- ▷ *Launch your preferred editor (emacs, vim, kate,...) in the background (using `&`)*
- ▷ *launch the python shell*

## 1 Object-Oriented Programming

A class is defined using the keyword `class`, followed by the class name and then "`(object)`". The class body must be indented, and the first expression can be a documentation string. A method is a function that "belongs to" a class. Methods must have, as their first parameter, the variable `self`, which contains a reference to the object itself. It is equivalent to the variable `this` in Java, with the difference that you need to pass it explicitly. By default, all methods and attributes in Python are public. It is possible to make them "pseudo" private, adding two underlines before their name (for example as in `__func(self)`), although dedicated hackers can still access private values if they wish.

▷ *Type the lines introduced by "`>>>`" and by "`...`"*

```
>>> class MyClass(object):
...     """A very simple class"""
...
...     def greet(self, name):
...         return "Hello, " + name
```

A class is instantiated into an object, and methods of the instance can be called as in Java.

▷ *Type the lines introduced by "`>>>`" and by "`...`"*

```
>>> c = MyClass() # object instantiation
>>> c.greet("paolo")
```

Classes can have an initialisation method `__init()__` similar to the class constructor in Java. This method is called when the class is instantiated and

can have a set of arguments. In contrast with Java, which requires different constructors to handle different types and numbers of arguments, Python uses only one constructor method per class.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> class Greeter(object):
...     """A simple class"""
...     def __init__(self, greeting):
...         self.greeting = greeting
...     def greet(self, name):
...         return self.greeting  + ", " + name
...
>>> c2 = Greeter("hi")
>>> c2.greet("tim")
```

The "`(object)`" after each class name in its definition specifies that it should derive (inherit) from the general-purpose "object" class. (Among other things, this makes the class be "new-style"; old-style Python classes inheriting from nothing are deprecated and should be avoided.) Classes can also derive from any other existing class or other classes, and they then inherit the methods and attributes of the parent class(es):

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> class GreeterEx(Greeter):
...     """A derived class"""
...     def bye(self):
...         return "Bye Bye"
...
>>> c3 = GreeterEx("hello")
>>> c3.greet("mr smith")

>>> c3.bye()
```

This class will contain the methods defined in `Greeter`, plus the new `bye()` method.

## Exercise 1

Using your preferred editor, create a class that checks if a string (`infix`) is contained in other strings. The class must be initialised passing the `infix` string, that must be stored in a class variable. The class must expose the method `check(string)` that verifies if `infix` is contained in the passed string (you can use the operator `in` to verify if a string is contained in another one: `string1 in string2`).*Remember to use* `self` *when trying to access methods and attributes of the instance.*

Import your module into the python shell, and test its behaviour (you must instantiate the class passing the infix string, and then call the method check passing different strings)

Note

*If you use the statement* `import modulename`*, remember to use the modulename prefix in front of the class name. If you make an error in the class, and you need to reimport the module, use* `reload(modulename)`*:* `import` *will not reimport a module already imported. You will also have to reinstantiate the class.*

# 2 More on Data Types

## 2.1 Lists

It is possible to create nested lists:

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> L1 = [1, 2, 3]
>>> L2 = ['one',L1,'two']
>>> L2
```

We have already seen the method `append()` for the list data type in the previous lab. We will see some more today. All the following operations are *in place*, that is they change the object on which they are applied.

▷ `insert(i,x)`
insert the item `x` at position `i`.
*Type:*

```
>>> L2 = ['a','b','d','e']
>>> L2.insert(1,'c')
>>> L2
```

▷ `remove(x)`
removes the first item in the list whose value is `x`.
*Type:*

```
>>> L2.remove('d')
>>> L2
```

▷ `pop([i])`
returns (and removes) the last item in the list (or the item in position `i`).
*Type:*

```
>>> L2.pop()
>>> L2
```

▷ `sort()`
sort the items of the list.
*Type:*

```
>>> L2.sort()
>>> L2
```

▷ `reverse()`
reverse the elements of the list.
*Type:*

```
>>> L2.reverse()
>>> L2
```

▷ `del`
can be used to remove items from a list using the index. It can also be used to remove slices from a list.
▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> del L2[1:3]
>>> L2
```

You can iterate over a list retrieving the index and the value at the same time using the `enumerate(list)` function.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> for i, v in enumerate(['a','b','c','d']):
...     print i, v
```

If you have two lists of the same length, you can step through both lists at once using the `zip(list1,list2)` function.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> for l,u in zip(['a','b','c','d'],['A','B','C','D']):
...     print l,u
```

**Exercise 2**

- ▷ Using your preferred editor, create a class named `Queue` that models a queue: the first element that enters is the first that exits (FIFO: First In, First Out). The class will use a list to maintain the data. It will expose the following methods:
  - ▷ `isempty()`: verifies if the queue is empty
  - ▷ `push(item)` inserts an element at the end of the queue
  - ▷ `pop()` : extracts and returns the first element in the queue (possibly only if the queue is not empty)
- ▷ Import the module into the python shell, and test it

*Remember to create the list that contains the data before accessing to it.*

**Exercise 3**

- ▷ Using your preferred editor, create a class named `Stack` that models a stack: the last element that enters is the first that exits (LIFO: Last in, First Out). The class willl use a list to maintain the data. It will expose the following methods:
  - ▷ `isempty()`: verifies if the stack is empty
  - ▷ `push(item)`: inserts an element at the end of the stack
  - ▷ `pop()`: extracts and returns the last element of the stack (possibly only if the stack is not empty)
- ▷ Import the module into the python shell, and test it

*Remember to create the list that contains the data before accessing to it.*

### Tuples

A *tuple* is composed of a number of values separated by commas, and enclosed by parentheses.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> T = (1,2,'three')
>>> T

>>> T[2]
```

Tuples can be nested.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> T1 = (1,2,(3,5))
>>> T1
```

Tuples (like strings and unlike lists) are immutable and can not bechanged once created. If you try, you will get an error message.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> s = "hello"
>>> s[1] = "u"

>>> T1[2] = 3
```

### Sets `(only from version 2.4)`

A set is an unordered collection with no duplicate elements. Set objects support mathematical operations like union, intersection, difference and symmetric difference.

*Type:*

```
>>> A = set([1,2,3,4,5,2,3])
>>> A

>>> B = [2,2,4,5,6,7]
>>> B

>>> 1 in A

>>> 1 in B

>>> A|B    # numbers in either A or in B (union)

>>> A&B    # numbers in A and B (intersection)

>>> A-B    # numbers in A but not in B
```

### Dictionaries

*Dictionaries* are indexed by keys, which can be any immutable objects (numbers, strings, tuples, etc.) Lists cannot be dictionary keys, because lists are mutable. A dictionary can be seen as an unordered set of *key:value* pairs, with

the constraint that keys need to be unique. The main operations performed on dictionaries are storing and retrieving values by their keys.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> num = {'one':1, 'two':2, 'three':3, 'four':4}
>>> num['three']

>>> num
```

You can easily add a new item to the dictionary:

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> num['five'] = 5
>>> num
```

You can delete one item from the dictionary using the built in function `del(item)`:

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> del(num['three'])
>>> num
```

To list all the keys from a dictionary, you use the method `keys()`. To check if a key belongs to the dictionary you use the method `has_key()`.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> num.has_key('one')

>>> num.keys()
```

You can iterate over a dictionary, retrieving the keys and their corresponding values using the method `iteritems()`

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> for k, v in num.iteritems():
...     print k,v
...
```

See also `iterkeys()` and `itervalues()`, which do what one might expect.

**EXERCISE 4**

▷ Create a class for managing a phone book. The user must be able to:

- ▷ insert a name and the associated phone number,
- ▷ obtain a number from a name,
- ▷ verify if name is in the book,
- ▷ list all the names and phone numbers in the book,
- ▷ delete a name from the book
- ▷ as *optional feature*, the user should be able to print in alphabetical order the names and their phone numbers

▷ Import your class into the python shell, and test it (remember to instantiate the class).

HINT
*Use a dictionary to store the data, and remember to create it before using it. You can use the method* `keys()` *to obtain the list of all the keys. Then you can apply any method available for the lists on the obtained list.*

# 3 Pattern matching

The `re` module provides a tools for regular expressions.

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> import re
```

▷ `match(pattern, string)`
If zero or more characters at the beginning of `string` match the regular expression `pattern`, it returns a corresponding `MatchObject` instance. It returns `None` if the string does not match the pattern.
*Type:*

```
>>> re.match("(aa|bb)+", "aabbaa")

>>> re.match("(aa|bb)+", "abba")
```

▷ `findall(pattern, string)`
It returns a list of all non-overlapping matches of `pattern` in `string`.
*Type:*

```
>>> re.findall("[a-z]*th[a-z]*", "I think this is the right one")
```

▷ `sub(pattern, repl, string)`
It returns the string obtained by replacing the leftmost non-overlapping occurrences of `pattern` in `string` by the replacement `repl`.
*Type:*

```
>>> re.sub("[a-z]*th[a-z]*", "TH-word", "I think this is the right one" )
```

**EXERCISE 5**

▷ Create a regular expression that checks if a string starts with 3 binary digits (and test it: `010asda` must be recognised, while `1aa` must be rejected)

▷ Using a regular expression, write a python statement that finds all the words that end with "`ly`" in strings (and test it, for example using the sentence "`it is likely to happen rarely`")

▷ Using a regular expression, write a python statement that replaces all the words that start with "`wh`" with "`WH-word`" (and test it, for example in the sentence "`who should do what?`")

# 4 Passing parameters to scripts

It is possible to pass parameters to a script.

- ▷ *exit from the python shell*
- ▷ *create in your editor a file named* `test.py`
- ▷ *Type in the editor:*

```
import sys

for arg in sys.argv:
    print arg
```

- ▷ *Save the file*
- ▷ *Type in the shell:*

```
python test.py these are the arguments
```

The arguments are stored in the variable `sys.argv`, which is a list of strings. `sys.argv[0]` contains the name of the script, while the following elements contain the arguments.

# 5 Advanced Topics

## 5.1 Class vs. instance attributes

When you define an attribute "x" in a Python class, it matters whether you set its value in the class definition or in the constructor for the class.

For instance,

```
>>>
>>> class A:
...     list1=[0,1,2]
...     def __init__ (self):
...        self.list2=[3,4,5]
...
>>>
>>> x=A()
>>> y=A()
>>> x.list1.append(6)
>>> x.list1

>>> x.list2.append(7)
>>> x.list2

>>> y.list1

>>> y.list2
```

That is, there is a copy *per instance* for anything set in the constructor, but only one copy *per class* for anything set in the class definition. This difference is subtle, but often crucially important.

EXERCISE 6

▷ Write a class that counts the number of times it has been instantiated. If an instance is deleted the counter is not decreased.

## 5.2 Memory management/argument passing

Like most scripting-style languages, Python does automatic memory management, and so the user does not normally need to worry about allocating and releasing system memory for objects. Basically, variables such as "x" or "cl.x" refer to objects stored in memory, and the system keeps track of how many references there are to each object. When there are no references remaining, Python will eventually remove the object from memory.

However, it is important to understand how objects in memory are named and passed around, particularly when arguments are given to classes and functions. For instance, after typing "x=0", the attribute "x" has the value 0, stored in some particular location in memory to which "x" points. If you then type "x=2", a *new object* with the value 2 is created, and then "x" is changed to point to that object instead. Note that this is very different from languages like C and Java, where "x=2" puts the value 2 into the location in memory to which "x" already points, rather than changing where "x" is pointing.

Thus, when calling a function, Python typically acts as if it uses 'call by value' semantics (even though it is really always passing references around):

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> x=0
>>> def fun(a): a=3 ; return a
...
>>> fun(x)

>>> x

>>>
```

Here "a" at first refers to the same item in memory that x does, but then after "a=3", "a" refers to a new object with the value "3". In any case, the value of "x" is still 0; `fun` cannot change what "x" points to.

Where this gets confusing is when "x" is what is called a "mutable" object, such as a list. For instance:

▷ *Type the lines introduced by ">>>" and by "..."*

```
>>> x=[0,1,2]
>>> def fun(a): a.append(3) ; return a
...
>>> fun(x)
```

```
>>> x

>>>
```

Here, a.append() modifies the same item in memory as "x" points to, because the append() operator modifies the item 'in place'. (Unlike the = assignment operator, which changes x to point to a new object!) Thus these results are very different from what is obtained when an assignment operator is used instead of append():

```
>>> x=[0,1,2]
>>> def fun(a): a = x+[3] ; return a
...
>>> fun(x)


>>> x

>>>
```

The simple rules to remember are that assignment with = changes the attribute to point to a new object (which has no effect if the attribute is just a function argument), but that otherwise objects *may* be modified in place, and it is important to know which functions or methods do that.