Python and Statistics for Matlab Users

Session 5

Nicolas Barrier

OT-Med Labex

19 septembre 2016

Table of Contents

Introduction to object programming

Object-oriented programming is a programming paradigm based on the concept of *objects*, which may contain data, in the form of fields, often known as attributes ; and code, in the form of procedures, often known as methods (source : Wikipedia)

```
import pylab as plt

# creation of a "Figure" object
fig = plt.figure()

print type(fig)  # <class 'matplotlib.figure.Figure'>
print fig.dpi  # "dpi" attribute assoc. with "Figure" object
plt.close(fig)  # "close" method assoc. with matplotlib
```

Introduction to object programming

Warning !

A function must be used on the right object !

```
# creation of a "file" object
fin = open("toto.txt", "w")
fin.close()   #  "close" method assoc. with "file" object
# no problems

fin = open("toto.txt", "w")
plt.close(fin)  # "close" method assoc. with matplotlib
# TypeError: Unrecognized argument type <type 'file'> to close
```

Conclusion

This is why it is unadvised to use wildcard imports :

```
from library import *
```

Class definition

```
class Names(object):  # defining Names class

    # class initialisation method
    def __init__(self, name1, name2):
        self.name1 = name1  # defining name1 attr.
        self.name2 = name2  # defining name2 attr.

    # redefines the output of the print/str function
    def __str__(self):
        return 'This is the new str method ' + \
            'with %s and %s' %(self.name1, self.name2)

    # method that returns a string. takes 1 arg.
    def hello_function(self, name3):
        return "%s and %s say hello " + \
            "to %s" %(self.name1, self.name2, name3)
```

Class definition : initialisation

```
class Names(object):

    def __init__(self, name1, name2):
        self.name1 = name1
        self.name2 = name2
```

In the `__init__` method, the first argument is `self`, which references the object being created. The initialisation is achieved as follows :

```
mynames = Names("Luc", "Paul") # 2 arguments, not 3!
mynames.name1 # recovering the name1 attr. ("Luc")
mynames.name2 # recovering the name2 attr. ("Paul")
```

Class definition : print/str output

When calling the `print` or `str` functions on an object, the `__str__` method associated with the object is called :

```
def __str__(self):
    return 'This is the new str method ' + \
        'with %s and %s' %(self.name1, self.name2)
```

Note

The `__str__` function does not take `name1` and `name2` as *argument*. It uses the `name1` and `name2` *attributes*.

```
print mynames
strout = str(mynames)
# This is the new str method with Luc and Paul
```

Class definition : methods

```
    # method that returns a string. takes 1 arg.
    def hello_function(self, name3):
        return "%s and %s say hello " + \
            "to %s" %(self.name1, self.name2, name3)
```

To call the method :

```
strout = mynames.hello_function("Michel")
# Luc and Paul say hello to Michel
```

Class inheritance

Object programming allows class inheritance.

Initialisation using the super statement :

```
class NewNames(Names): # "object" replaced by Names

    def __init__(self, name1, name2, name3):

        # Initialisation through the mother class
        super(NewNames, self).__init__(name1, name2)

        self.name3 = name3
```

super(NewNames, self) is the instance of the mother class. Hence, the __init__ method of the mother class is first called. Then we add the name3 attribute.

The class is initialised as follows :

```
mynewnames = NewNames("Luc", "Paul", "Raymond")
```

Class inheritance

The `hello_function` method of the mother class remains accessible :

```
newnames.hello_function("Michel")
# Luc and Paul say hello to Michel
```

You can also add methods :

```
    def new_hello_function(self):
        return "%s and %s say hello to %s"\
            %(self.name1, self.name2, self.name3)
```

The `new_hello_function` method is called as follows :

```
newnames.new_hello_function()
# Luc and Paul say hello to Raymond
```

Note

Remark the absence of arguments in the call to the `new_hello_function()`.

Class inheritance

Inheritance allows to overwrite a method associated with an object (as done in the above with the `__str__` method).

```
class FinalNames(NewNames):

    def __init__(self, name1, name2, name3):
        super(FinalNames, self).__init__(name1, name2, name3)

    # overwritting of the NewNames
    # new_hello_function method
    def new_hello_function(self):
        output = "%s and %s say NEW HELLO to %s" \
            %(self.name1, self.name2, self.name3)
        return output

finnames = FinalNames("Luc", "Paul", "Raymond")
finnames.new_hello_function()
# Luc and Paul say NEW HELLO to Raymond
```

Table of Contents

Run with arguments

Arguments are extracted by using the sys module :

```
import sys

sys.argv
# list of string arguments
# sys.argv[0]: script name
# sys.argv[1]: first argument
```

To run the program :

```
python my_program.py arg1 arg2 ...
```

Exectuable program

To make a program executable, add on top of your python script the line :

```
#!/opt/local/bin/python
```

This is the path to your python executable.

Warning !

It must be consistent with the one you used when installing libraries

Then make your `.py` file an executable. In Linux/Mac Os X :

```
> chmod 755 my_program.py
```

To run the program :

```
> ./my_program.py arg1 arg2 ...
```

__main__ condition

Python has no `main()` function, as in C or Java. If you want to execute a part of a script only when it is called as the main program :

```
print "inside my_prog.py"

if __name__ == "__main__":
    print "my_prog.py is main"
```

```
> python
>> import my_prog
inside my_prog.py

> python my_prog.py
Inside my_prog.py
my_prog.py is main
```

Table of Contents

Try statement

The `try` statements allow to test the execution of a part of a code. If it fails, the user can choose to either pass or to raise an error.

```
x = 0

try:
    print x # try to print x
except:
    print "print x fails: raise error"
    raise # raise an error if the try fails
print 'ok'
# 0
# ok
```

Try statement

```
x = 0

try:
    print y # try to print y
except:
    print "print y fails: raise error"
    raise # raise an error if the try fails
print 'ok'

# print y fails: raise error
# Traceback (most recent call last):
# File "except.py", line 19, in <module>
#    print y
# NameError: name 'y' is not defined
```

Try statement : pass

```
x = 0

try:
    print y # try to print y
except:
    print "print y fails: pass"
    pass # skip if the try fails
print 'ok'

# print y fails: pass
# ok
```

Complicated try statement

```
try:
    fin = open("toto.txt", "r")
except ZeroDivisionError: # if fails because of div. by 0
    print "================= division by 0"
    raise
except IOError: # if fails because of an IO error
    print "================= cannot open file "
    raise
except: # if fails because of another error
    print '================= unknown error'
    raise

# ================= cannot open file
# Traceback (most recent call last):
# File "except_comp_1.py", line 2, in <module>
#   fin = open("toto.txt", "r")
# IOError: [Errno 2] No such file or directory: 'toto.txt'
```

Complicated try statement

```
try:
    0/0
except ZeroDivisionError: # if fails because of div. by 0
    print "================= division by 0"
    raise
except IOError: # if fails because of an IO error
    print "================= cannot open file "
    raise
except: # if fails because of another error
    print '================= unknown error'
    raise

# ================= division by 0
# Traceback (most recent call last):
# File "except_comp_2.py", line 2, in <module>
#     print 0/0
# ZeroDivisionError: integer division or modulo by zero
```

Complicated try statement

```
try:
    print toto
except ZeroDivisionError: # if fails because of div. by 0
    print "================= division by 0"
    raise
except IOError: # if fails because of an IO error
    print "================= cannot open file "
    raise
except: # if fails because of another error
    print '================= unknown error'
    raise

# ================= unknown error
# Traceback (most recent call last):
# File "except_comp_3.py", line 2, in <module>
#     print toto
# NameError: name 'toto' is not defined
```