Introduction to programming.

Lecture 5: Modules and Class

Lecturers: Giovanni Casini (giovanni.casini@uni.lu) and Xavier Parent (xavier.parent@uni.lu)

Adapted from the original materials by Clément Guérin

Modules in Python

A **module** is a piece of code stored in a file *.py that you can **import** and include in your application.

Why not to write everything in the main code? There are many possible reasons.

For example, the use of modules can:

- ease maintanance and optimisation
- allow a better organisation of the code for your application
- allow to resuse important pieces of code (e.g. important functions) just by importing them
- ease collaborative work
- use already existing pieces of code

In principle, any .py file can by used as a module.

Given a file ***NameFile.py***, it can be used as a module, that is automatically named ***NameFile***

It can then be called by using the command **import**

General methods to use import

There are two different ways to use the **import** command on a module ***NameFile.py***.

Only import the module

import *NameModule*

If you need only a function ***function*** defined within the code of the module ***NameModule***, you will need to call

NameModule.function.

In order to use your module in Spyder:

- Create the file ***NameModule.py*** with your module
- Create a main file that imports the module (command **import *NameModule***)
- Put both the file in the same folder
- Run the module in Spyder (don't forget it!)
- Run the main file

Python has many built-in modules. Today we will consider some of them.

In [4]:

```
#example that we already saw
import random
print(random.randrange(100))
```

Out[4]:

```
47
```

In [5]:

```
import numpy
print(numpy.pi) #Here numpy.pi denotes an approximation of the usual pi number
```

Out[5]:

```
3.141592653589793
```

Once the line of command "**import *NameModule***" is read, ***NameModule*** should be thought of as a variable whose type is **module**.

Functions defined in ***NameModule*** are then called **as if they were methods for *NameModule***.

Each time you call them, the interpretor will go through the module file and read the behavior of the object you just called.

In [9]:

```
print(type(random)) # Type : module
print(type(random.randrange)) # Type : method
```

```
<class 'module'>
<class 'method'>
```

It is possible to change the name of your module for another name (usually shorter).

import *NameModule* as *NewNameModule*

The application will recognise the method with the new name. If you use the old name, it will give you error.

In [2]:

```
import math as m # math is not defined and
                 # m is the object containing functions of the module math
print(m.pi) # It works
math.pi #NameError
```

```
3.141592653589793


---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-2-ee71087e1d9d> in <module>
      2                  # m is the object containing functions of t
he module math
      3 print(m.pi) # It works
----> 4 math.pi #NameError

NameError: name 'math' is not defined
```

Import objects from the module

The other way to import tools from modules is to directly redefine functions, constants... from modules.

from *modulename* **import** *this*, *that* **as** *name*1, *name*2

In [3]:

```
from math import pi as sliceofpie # Now sliceofpie is a floating number with the
value math.pi
print(sliceofpie)
# ============================================================================
from math import pi # Of course the "as" command is optional
print(pi==sliceofpie) # It is the same
```

```
3.141592653589793
True
```

In [3]:

```
from math import pi as sliceofpie # Now sliceofpie is a floating number with the
value math.pi
print(sliceofpie)
# ============================================================================
from math import pi # Of course the "as" command is optional
print(pi==sliceofpie) # It is the same
```

```
3.141592653589793
True
```

In [3]:

```
import random
print(randrange(100)) #error: must be used as a method
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-3-ca021cbb8c0f> in <module>
      1 import random
----> 2 print(randrange(100)) #error: must be used as a method

NameError: name 'randrange' is not defined
```

In [4]:

```
import random
print(random.randrange(100))
```

```
48
```

In [5]:

```
from random import randrange
print(randrange(100)) #no error: using From "method " import "fuction"
                      #allows to use directly functions instead of methods
```

```
84
```

You can import everything from a module by writing

from *modulename* **import** $*$

It is very similar to

In [6]:

```
import math
c = factorial(3)
print(c)
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-6-dd79da97b581> in <module>
      1 import math
----> 2 c = factorial(3)
      3 print(c)

NameError: name 'factorial' is not defined
```

In [7]:

```
import math
c = math.factorial(3)
print(c)
```

```
6
```

In [8]:

```
from math import *
c=factorial(3) #factorial is a function in the module math
print(c)
```

```
6
```

Some of the most used built-in modules

If you downloaded the Anaconda distribution, the modules that we are going to talk about can be directly imported.

No need to download a specific library from the web.

Module math

For most used modules, the Python documentation is well written https://docs.python.org/3.6/library/math.html (https://docs.python.org/3.6/library/math.html).

The math library contains most usual mathematical functions (log, exp, trigonometric functions,...) and constants (pi,e,...).

In [5]:

```
from math import inf,e

print(inf+inf)
print(1/inf)
print(inf-inf)# Undetermined form works even though
print(e)
```

```
inf
0.0
nan
2.718281828459045
```

In [9]:

```
from math import log
log(4) #without a second argument, it is assumed that e is the base
```

Out[9]:

```
1.3862943611198906
```

In [14]:

```
import math
math.log(4)
```

Out[14]:

```
1.3862943611198906
```

In [15]:

```
from math import log
log(4,10) #the optional second argument specifies the base
```

Out[15]:

```
0.6020599913279623
```

Module numpy

Documentation is here : http://www.numpy.org (http://www.numpy.org). Particularly interesting when dealing with linear algebra.

You first have the numpy "array".

In [20]:

```
import numpy as np
```

In [21]:

```
v=np.array([[1,1],[1,1]])# Create a 2 by 2 array
print(v)
print([[1,1],[1,1]])
```

```
[[1 1]
 [1 1]]
[[1, 1], [1, 1]]
```

In [22]:

```
v+=v
print(v)
```

```
[[2 2]
 [2 2]]
```

In [23]:

```
v=np.array([[1,2],[3,4]])
print(v)
print(v[0][1]) # You can access the elements of an array
v[0][1]=5 # You can reassign a value of an array
print(v)
w=v     # Arrays are mutable objects
w[0][1]=3
print(w,'\n',v)
```

```
[[1 2]
 [3 4]]
2
[[1 5]
 [3 4]]
[[1 3]
 [3 4]]
 [[1 3]
 [3 4]]
```

In [24]:

```
np.array([[True,2.0],[0+1j,0.1],(3+7*1j,-np.pi)]) # The array function will unif
ormize the type
                                                  # you entered.
```

Out[24]:

```
array([[ 1.00000000+0.j,  2.00000000+0.j],
       [ 0.00000000+1.j,  0.10000000+0.j],
       [ 3.00000000+7.j, -3.14159265+0.j]])
```

You also have a type **Matrix**.

In [18]:

```
M=np.matrix([[0,1j],[1j,0]]) # Type : matrix
print(M.conjugate()) # Returns the conjugate of a matrix
print(M*M)
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-18-7d4c24790f17> in <module>
----> 1 M=np.matrix([[0,1j],[1j,0]]) # Type : matrix
      2 print(M.conjugate()) # Returns the conjugate of a matrix
      3 print(M*M)

NameError: name 'np' is not defined
```

In [27]:

```
D=np.matrix([[0,3],[5,5]])
N=D    # Matrix are mutable objects
N[0,1]=2713
print(D)
```

```
[[    0.+0.j  2713.+0.j]
 [    0.+1.j     0.+0.j]]
```

Module scipy

Documentation is here : https://www.scipy.org (https://www.scipy.org). Contains numpy (numerical analysis), sympy (symbolic computation) and other features that we will check out within several weeks.

Module Random

As its name indicates, use it to randomize your script.

Documentation: https://docs.python.org/2/library/random.html (https://docs.python.org/2/library/random.html)

In [6]:

```
import random
c=random.random() # Returns a random float between 0 and 1
print(c)
```

```
0.8988967536406869
```

In [31]:

```
c=random.randrange(0,4) # Return a random integer between 0 and 3 (included)
print(c)
```

Out[31]:

```
2
```

In [32]:

```
Months=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
random.shuffle(Months) #Mixes up the list Months (Months is changed)
print(Months)
```

Out[32]:

```
['Sep',
 'Dec',
 'Aug',
 'Jul',
 'Oct',
 'Apr',
 'Feb',
 'Mar',
 'Jun',
 'May',
 'Nov',
 'Jan']
```

In [9]:

```
Months=['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sep','Oct','Nov','Dec']
random.sample(Months,k=5) # Sample of 5 elements in the population Months
#Compare random.shuffle(L) and random.sample(L,len(L))
```

Out[9]:

```
['Jun', 'Mar', 'Jan', 'Apr', 'Oct']
```

Exercise

Let us throw 10 dices with 1,2,3,4,5,6.

- You win 2€ if the sum of the values on the dice is from 10 to 23.
- You loose 160€ if the sum of the values on the dice is either 24 or 48.
- You win 250€ if the sum of the values on the dice is 42.

Should you play the game? Give an estimation of your average winnings/losts considering 1000 tests.

Classes in Python

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Definition

In [23]:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

```
  File "<ipython-input-23-800015dc0bd8>", line 2
    <statement-1>
    ^
SyntaxError: invalid syntax
```

In [34]:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'


print(MyClass.i)


a=MyClass()
print(MyClass.f(a))
```

```
12345
hello world
```

The init() Function

To understand the meaning of classes we have to understand the built-in **init**() function.

All classes have a function called **init**(), which is always executed when the class is being instantiated.

Use the **init**() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

In [36]:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

d = Dog('Fido')
e = Dog('Buddy')
d.kind                  # shared by all dogs
e.kind                  # shared by all dogs
d.name                  # unique to d
e.name                  # unique to e
```

Out[36]:

```
'Buddy'
```

In [1]:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

```
John
36
```

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

In [2]:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

Once you have created an object with certain property values, you can:

- Modify the value of a property
- Delete a property
- Delete the object itself

In [ ]:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print(abc.age)

p1 = Person("John", 36)
p1.myfunc()
p1.age = 40
p1.myfunc()
```

In [ ]:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print(abc.age)

p1 = Person("John", 36)
p1.myfunc()
del p1.age
p1.myfunc()
```

In [ ]:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print(abc.age)

p1 = Person("John", 36)
del p1
p1.myfunc()
```

In [39]:

```
class Ratio():
    "rational number"
    def __init__(self,numerator,denominator): #Always use "self" as a first vari
able
        self.num=numerator
        self.den=denominator
```

In [40]:

```
q=Ratio(0,1) # Create a variable "q" of type  "Ratio" whose "num" attribute is 0
and "den" attribute is 1
print(q.num)
print(q.den)
```

```
0
1
```