2_Functions_Classes_Modules

February 14, 2020

1 Functions

In programming, a function is a named section of a program that performs a specific task.

1.1 How to create a function?

In python we use the `def` statement to define a new function and place the content of the function in an **indented block**.

[ ]:
```
def myfunction():
    print('This is dummy')
```

[ ]:
```
# call of the function
myfunction()
```

[ ]:
```
def myadd(myparam1, myparam2):
    print('my first parameter is %s' % myparam1)
    print('my second parameter is %s' % myparam2)
    return myparam1 + myparam2
```

[ ]:
```
#Call of a function
myadd(12, 6)
```

Note: a function always returns a value, if no `return` statement is define it will return `None` by default

1.2 Function parameters:

- The function signature defines a set of parameters to be given when calling a function.
- Default values for a parameter can be given after an = sign
- There is more options for function parameters (like defining arbitrary argument list) that won't be discuss today but they are presented in the official python documentation or in pythoncentral tutorial

[ ]:
```
def myfunction(myparam1, myparam2=5):
    """This function simply print the given parameter"""
    print('my parameters are %s and %s' % (myparam1, myparam2))


myfunction(1)
myfunction(2, "toto")
```

```
myfunction(3, myparam2="titi")
```

1.3 Documentation of the function

One should use a **doc-string**, i.e. a string defining what the function does:

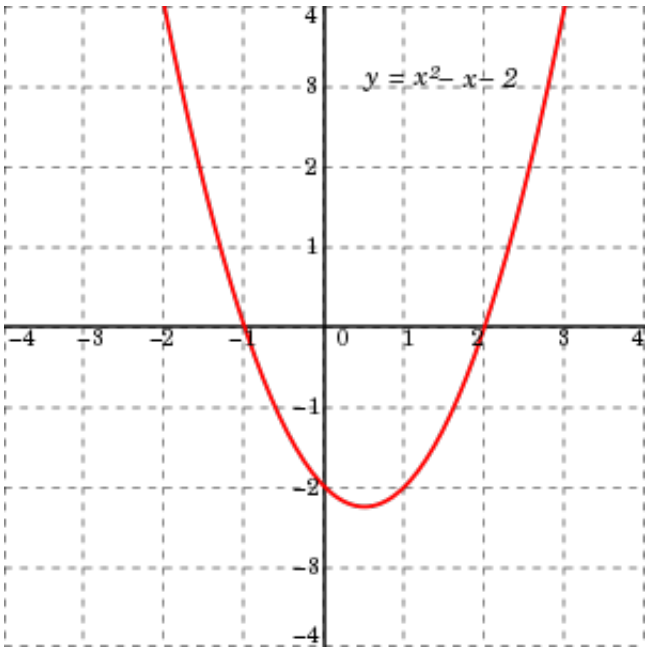
[ ]:
```
def myadd(myparam1, myparam2):
    "return the addition of the two input parameters"
    print('my first parameter is %s' % myparam1)
    print('my second parameter is %s' % myparam2)
    return myparam1 + myparam2

help(myadd)
```

You can automatically generate html, pdf, latex ... documentations from those docstrings. This part is presented in the software engineering training.

1.4 Hands on:

Write a function solving the quadratic equation: $a \cdot x^2 + b \cdot x + c = 0$.



quadratic_eq_example

note: For the quadratic function $y = x^2x2$, the points where the graph crosses the x-axis, $x = 1$ and $x = 2$, are the solutions of the quadratic equation $x^2x2 = 0$.

This function will take a, b and c in input and return the list of solutions (in **R**) for:

$$a \cdot x^2 + b \cdot x + c = 0$$

Reminder:

$$\Delta = b^2 - 4 \cdot a \cdot c$$

if $\Delta > 0$ then the equation has **two solutions**

$$\frac{-b - \sqrt{\Delta}}{2a}$$

and

$$\frac{-b + \sqrt{\Delta}}{2a}$$

if $\Delta = 0$ the the equation has **one solution**

$$\frac{-b}{2a}$$

if $\Delta < 0$ then there is **no solution** in **R**

Nota: square root of x can be obtained by x**(0.5)

```
def sqrt(x):
    """returns square root of x"""
    return x**(0.5)
```

[ ]:
```
# solution
import inspect
from solution_quadratic_function import polynom
print('Solution:')
print(inspect.getsource(polynom))
```

[ ]:
```
#example of usage:
polynom(1,-1,-2)
```

1.5 Warning about default mutable objects:

warning

Never use mutable objects as default parameter, or you will experience trouble !!!

If the parameter is a mutable, its default value should generally be None (immutable) and initialize an empty container.

Example:

[ ]:
```
def bad_append(any_list=[]):
    """Append 1 to provided list and return it.
    If no list is given as parameter, use empty list."""
    any_list.append(1)
    return any_list


print(bad_append())
```

[ ]:
```
print(bad_append())
```

1.5.1 Solution

The default value should generally be `None` which is immutable and initialize an empty container if needed.

[ ]:
```
def good_append(any_list=None):
    if any_list is None:
```

```
        any_list = []
    any_list.append(1)
    return any_list

print(good_append())
print(good_append())
print(good_append())
print(good_append())
```

1.6 Lambda function

One can define **anonymous** function, sometimes called lambda function in functional programing languages.

Nota: We don't expect you to use lambda, but this is just to explain why you can get an error when trying to use a variable called `lambda`, as it is a reserved keyword.

[ ]:
```
lambda = 1.3e-10
```

```
pow2 = lambda x: x*x
pow2(5)
```

2 Classes

Classes are used for Object Oriented Programming, they are **out of the scope of this training**.

2.1 Definition

They are defined by the `class` keyword to define the block corresponding to the class definition. The parameter `self` passed as first argument of any method is used to retrirve the instance of the class. The constructor method is called `__init__`, there is usually no destructor as objects are *garbage collected* in Python.

Here is just a simple example:

[ ]:
```
class MyClass(object):
    "Simple class inheriting from object"
    def __init__(self, param):
        "Constructor method"
        object.__init__(self)
        self.param = param

    def mymethod(self):
        print('value of my param is: %s'% self.param)
```

2.2 Instantiation

Instantiation is the creation of an object of a given class.

[ ]:
```
# creation of a new class instance
c = MyClass(2)

# access to a class method
c.mymethod()
```

[ ]:
```
# access to a class attribute
c.param
```

[ ]:
```
# check the class of an object
isinstance(c, MyClass)
```

3 Modules

A module is:

- a library containing useful variables, functions and classes to be used from different places (python script, python interactive interpreter…).
- a simple text file with the `.py` extension. !!! each module should group specific functionnalities. !!!

  a good practice is to specify the coding in each new python file 'header' with the following line to specify the file encoding

  ```
  # coding: utf-8
  ```

3.1 Example of a module:

```python
#!/usr/bin/env python
# coding: utf-8

"""This is a simple demonstration library"""

__authors__ = ["Pierre Knobel", "Jerome Kieffer", "Pierre Paleo", "Henri Payno",
               "Armando Sole", "Valentin Valls", "Thomas Vincent"]
__date__ = "17/11/2016"
__license__ = "MIT"


version = '0.1.0'


def sqrt(x):
    "Return the square root of x"
    return x**0.5


def polynom(a, b, c):
    """Compute the polygon of order two

    :param a:a value of the polynom
    :type a: float
    :param b: b value of the polynom
    :type b: float
    :param c: c value of the polynom
    :type c: float
    """
    delta = pow2(b) - 4.0 * a * c
    solutions = []
    if delta > 0:
        solutions.append((-b + sqrt(delta)) / (2.0 * a))
        solutions.append((-b - sqrt(delta)) / (2.0 * a))
    elif delta == 0:
        solutions.append(-b/(2.0*a))
    return solutions


def pow2(x):
    """

    :param x: input value
    :type x: float
    :return: the square of x
    """
    return x*x


def test():
    """Test the library"""
    assert(pow2(2.0) == 4.0)
    print("All OK")


if __name__ == '__main__':
    print("Running unit tests")
    test()
```

Example of module

3.2 How to import the module:

There is many ways to import modules:

[ ]:
```
import mymodule
mymodule.pow2(5)
```

[ ]:
```
import mymodule as mm
mm.pow2(6)
```

[ ]:
```
from mymodule import pow2
pow2(7)
```

[ ]:
```
#You can also access to the attributes of the module:
mymodule.__authors__
```

[ ]:
```
mymodule.version
```

[ ]:
```
help(mymodule)
```

3.3 Executing a module as a script

You can run a python module with:

`python mymodule.py <arguments>`

The code in the module will be executed as if you imported it.
The difference is that in this case if you have a **main** section it will be executed as well.

- A **main** section is defined with `if __name__ == "__main__":`

```
if __name__ == "__main__":
    print('Running unit tests')
    test()
```

If we execute the file:

```
$ python mymodule.py
Running unit tests
All OK
```

3.3.1 Make a python module file executable on Unix.

For this you will need:

- a main section
- Specify the name of the interpretor on first line of the file like `#!/usr/bin/env python`
- Make the script executable using `chmod +x filename`

official python documentation for executing modules: https://docs.python.org/3.7/tutorial/modules.html#executing-modules-as-scripts

If we execute the file:

```
$ ./mymodule.py
Running unit tests
All OK
```

3.4 Exercise

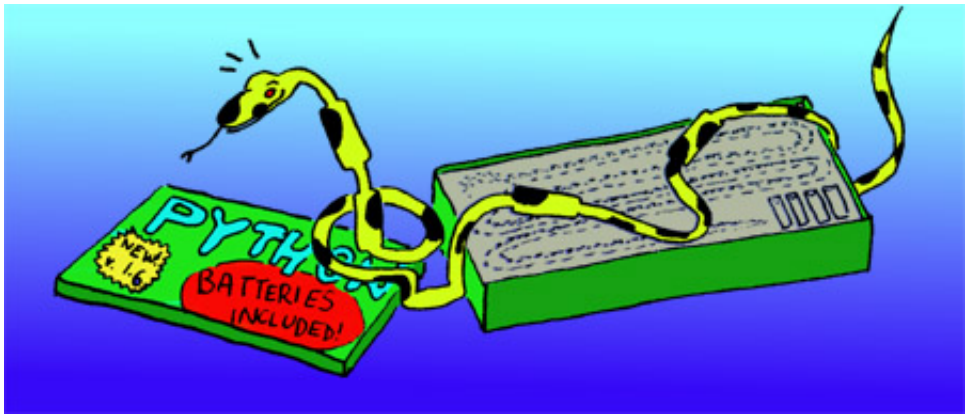
0. create a new file exercise.py with the file encoding and a description of the module
1. add a function into this file like polynom(a, b, c) defined previously and write a test() function to the first function is working.
2. import this module from a python console (i.e from the notebook . . .).
3. execute the function you wrote from the console and run the test.
4. execute this file as a script:

- defining the 'main' section:

```
if __name__ == '__main__':
    # operations to be executed
```

- execute python3 exercise.py

3.5 Standard modules



Batteries included philosophy

- Modules sys, os, shutil, glob, copy
- Modules string, re, collections
- Modules math, random, decimal
- Module time, datetime
- Internet access with email, urllib2, smtplib
- Multi-core programming with multiprocessing, threading, thread
- Handle compressed archives with gzip, bz2, zlib, zipfile, tarfile
- Execute another program with subprocess, shlex
- Quality control with unittest and doctest
- Performance control with timeit, profile and cProfile
- Logging capabilities: logging

3.6 Non standard modules

- General purpose mathematics libraries:
  - NumPy
  - SciPy
- Input/Output libraries to handle data acquired at ESRF
  - Silx
  - FabIO
  - H5py
- Visualization libraries (curves, images, . . .)
  - Matplotlib
  - Silx
- Image handling library:
  - Python Imaging Library (PIL Pillow)

They will be introduced this afternoon.

4 Errors and Exceptions

4.1 Syntax Errors

When you will start python you will certainly encounters several (Syntax errors). Those are checked before actual execution.

[ ]:
```
if True
    print(True)
```

4.2 Exceptions

4.2.1 Definition

Any other error than Syntax Errors are called exceptions: * ZeroDivisionError * NameError * TypeError * . . .

[ ]:
```
1/0
```

[ ]:
```
undef_var + 1
```

[ ]:
```
'2' + 8
```

4.2.2 Exceptions type:

- Plenty of exceptions are available (ImportError, RuntimeError, . . .) see built-in exceptions
- You can create your own exceptions
  - Need to create a new class
  - This is out of the scope of this training

4.2.3 Raising an exception:

The `raise` statement allows the programmer to force a specified exception to occur.

[ ]:
```
raise Exception('My personal message')
```

[ ]:
```
raise ValueError('-1 is not an unsigned int')
```

4.2.4 Handling exceptions

In some cases you might wan't to handle raised exceptions.

For this we will use the `try` statement like:

```
try:
    # ... some code that may break
except(TypeError, ExceptionXXX...):
    # ... what to do if those exception appears
else:
    # executed if no error found
finally:
    # always executed (i.e. to close file)
```

[ ]:
```
try:
    raise ValueError('Invalid value')
    print('this code will not be executed if an error is raised before')
except ValueError as e:
    print('an error appears:', e)
else:
    print('this code will be executed if no error raised')
finally:
    print('this will always be executed at the end')
```

[ ]:
```
# complete example: code that breaks from time to time:

def unreliable(t):
    d = int(t) % 2
    try:
        res = t/d # may divide by zero
    except ZeroDivisionError as e:
        print("Division by zero is not a good idea")
        raise RuntimeError("time is even !")
    else:
        print("This time everything went smoothly")
    finally:
        print("It is time to wrap-up")

import time
unreliable(time.time())
```

[ ]:
```
unreliable(time.time())
```