Python Tutorial

August 22, 2023

Contents

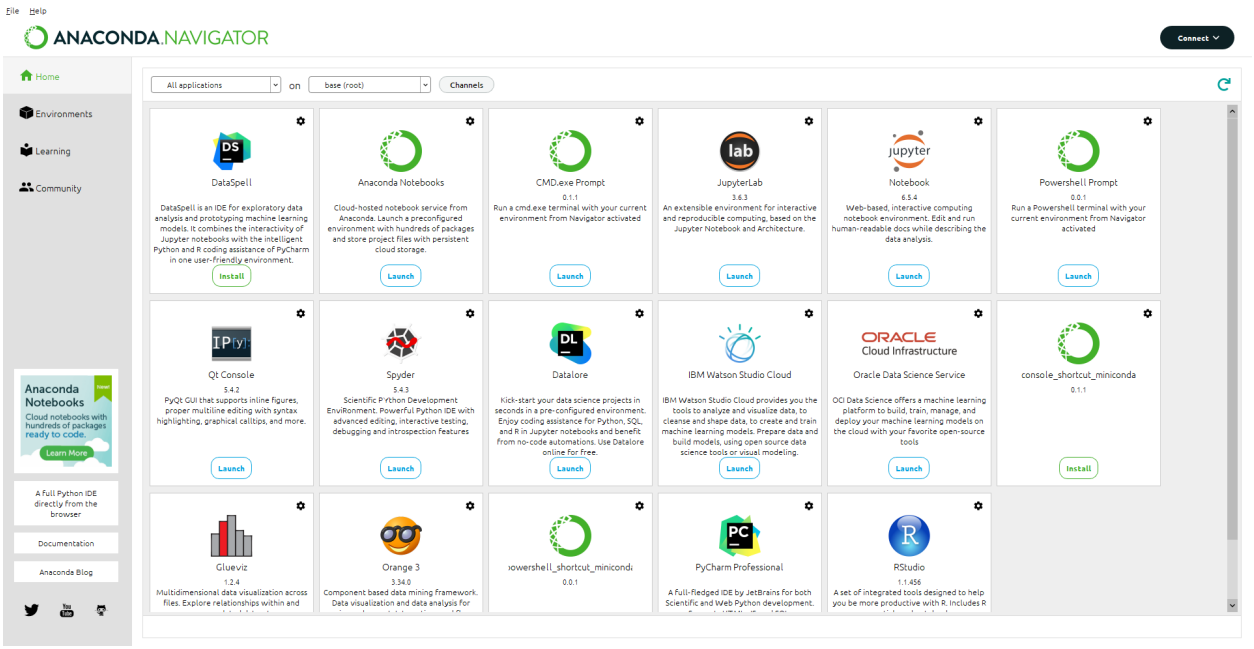
1 Installing Anaconda, Python, and the Spyder IDE

Python is a high-level, object-oriented open-source programming language widely used in the industrial and scientific world. This language not only allows for scientific computing and graphical visualization but also the creation of websites. To take advantage of it, we will use the Anaconda distribution platform which enables the installation of Python and the Spyder programming environment. To get started, download and install the Anaconda distribution for your operating system (Windows, Mac OS, or Linux):

https://www.anaconda.com/products/individual

By installing Anaconda, you will install Python, Jupyter Notebook, and Spyder. Once Anaconda is installed, you can launch the Spyder programming environment and run Notebooks (like this tutorial, for example).

After you have installed Anaconda, start the application. From the home screen, you can start either Spyder or Jupyter Notebook by double-clicking on the respective icon.



1.1 IDE Spyder

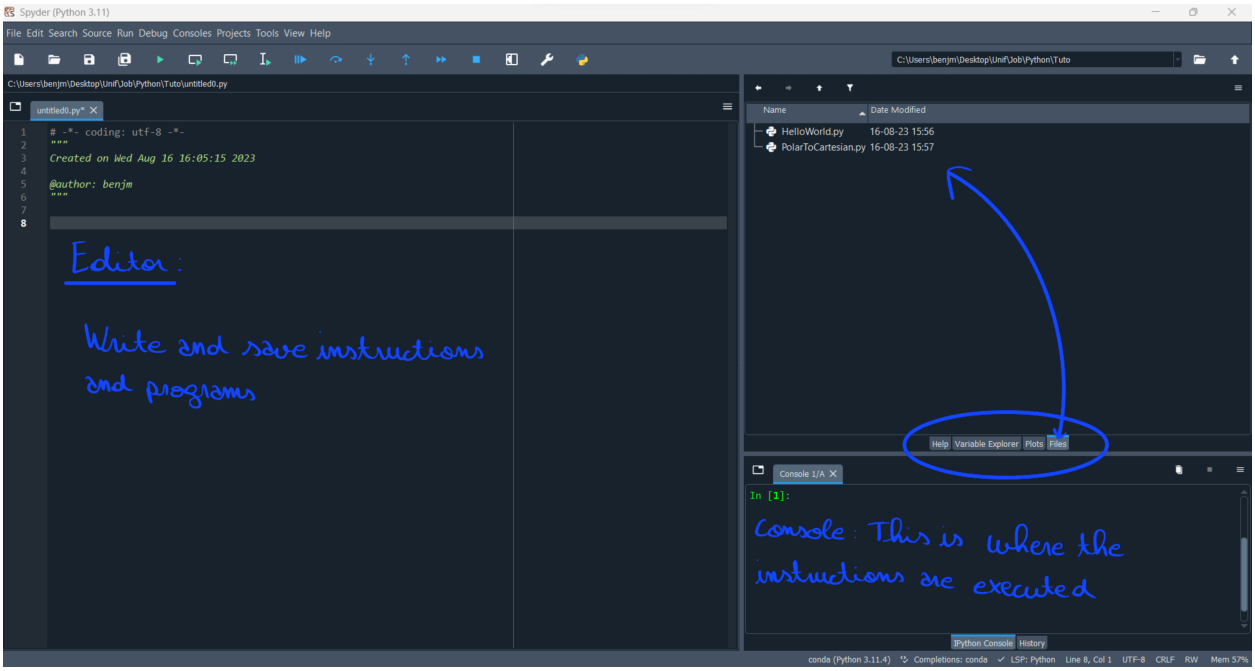
Spyder is the Scientific PYthon Development Environment: a powerful interactive development environment for the Python language with advanced editing, interactive testing, debugging, and performance analysis (profiler) capabilities.

https://www.spyder-ide.org/.

- In Spyder, the IPython console is the default Python interpreter.
- Code from the editor can be fully or partially executed in this console.
- The editor supports automatic error-checking for Python.
- The IPython debugger can be activated.
- A profiler is provided to analyze code efficiency.
- An object explorer displays documentation for functions, methods, etc.

- The variable explorer shows the names, sizes, and values of numerical variables.
- The file explorer allows you to navigate through your file tree.

Once Spyder is launched, you will see an interface divided into three main panels: the editor, where you can write and save your commands and programs; the file and figure explorer, along with the help section; and the console that executes the commands written in Python along with a history of entered commands. This last feature is very useful for tracing back in time and allows you to see which commands were executed, in what order, and at what time.



The IPython console allows you to enter commands after the prompt "`In [ ]:`".

```
In [1]: a=2
```

The example above shows that we assigned the value 2 to the variable `a` using the `=` operator. A variable allows to store data, whether as a scalar, an array, or a string. We can assign values to multiple variables at the same time. The assignment does not automatically show the value, so when we run the command `a=2`, the value of `a` does not appear. To see the value of `a`, we can use the `print()` function or the variable explorer. In the example below, we created a variable named `a`, a one-dimensional integer with a value of "2".

The variable explorer is our workspace. It provides information about the variables defined at that moment. It includes features for variable management, like creation, editing, and deletion. This information can also be accessed with the `who` and `whos` commands or the `type()` command.

```
In [2]: a1 = a2 = a3 = 3
        print(a1 * a2 * a3)
```

```
27
```

```
In [3]: print(a)
```

```
2
```

```
In [4]: who
```

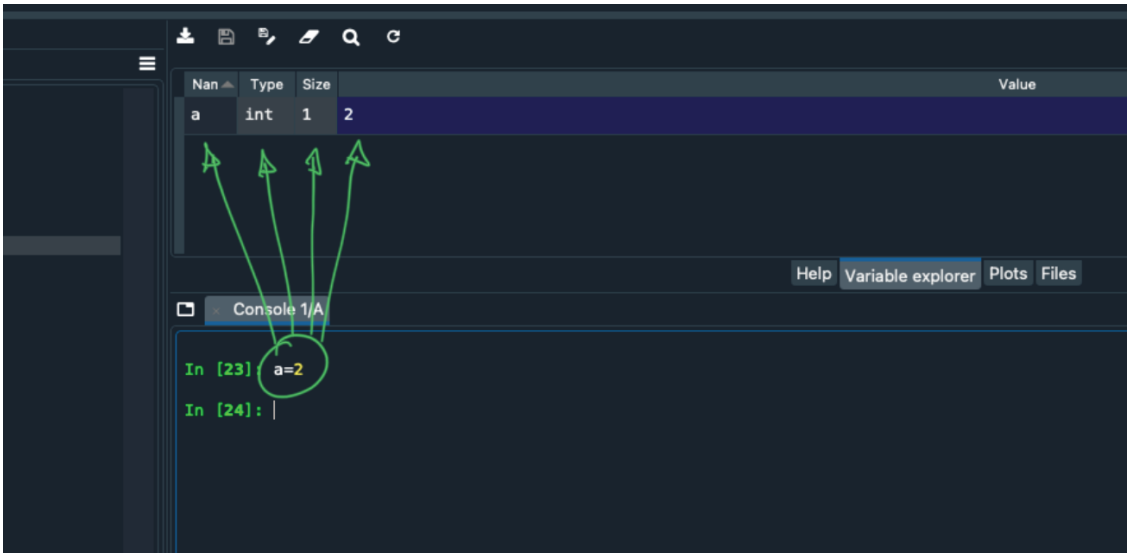
```
a        a1       a2       a3
```

```
In [5]: whos
```

```
Variable   Type    Data/Info
---------------------------
a          int     2
a1         int     3
a2         int     3
a3         int     3
```

```
In [6]: type(a)
```

```
Out [6]: int
```



If we want to delete variables, we can use the `%reset` command (which asks for confirmation) to clear all variables, or the `del` command to delete a specific variable. The `clear` command only cleans the console while keeping the variables in memory.

```
In [7]: del a
```

```
In [8]: %reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])?
Nothing done.
```

You can also use the addition `+`, subtraction `-`, multiplication `*`, division `/`, and power `**` operators or combine them. For example, `+=` increments the assigned variable by a specified value.

In [9]:
```
a = 2 + 3
print(a)
```

```
5
```

In [10]:
```
print(2**3)
```

```
8
```

In [11]:
```
a += 1
print(a)
```

```
6
```

In [12]:
```
a -= 1
print(a)
```

```
5
```

Python defines several basic operations by default (assignment, arithmetic, power, absolute value, comparisons, logic):

In [13]:
```
1 != 2
```

Out [13]: True

In [14]:
```
True & False
```

Out [14]: False

In [15]:
```
True & True
```

Out [15]: True

2 Variable Types

In Python, it is not necessary to declare variables before using them. The value assigned to the variable defines its type. The main types are:

Type `int` (integer)

In [16]:
```
a = 300
type(a)
```

Out [16]: int

Type `float` (floating-point number)

In [17]:
```
a = 1.25e3
type(a)
```

Out [17]: float

Type `complex` (complex number)

In [18]:
```
a = 1 + 3j
type(a)
```

Out [18]: complex

Type `str` (string)

In [19]:
```
a = "hello"
print(a)
type(a)
```

```
hello
```

Out [19]: str

In [20]:
```
a = 'goodbye'
print(a)
type(a)
```

```
goodbye
```

Out [20]: str

Type `bool` (booleans)

In [21]:
```
a = True
type(a)
```

Out [21]: bool

In [22]:
```
b = not(a)
print(b)
```

```
False
```

We used the `not()` operator to return the opposite of the provided boolean value.

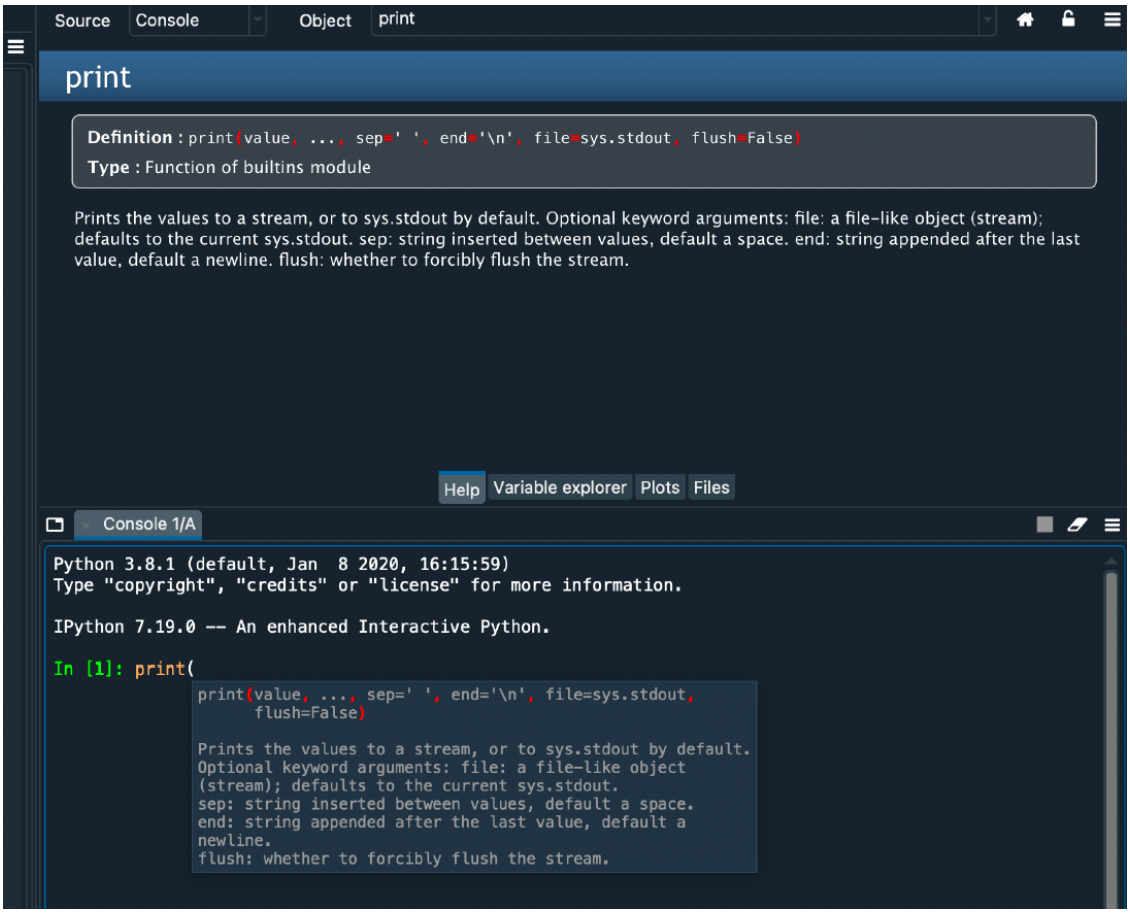
Exercise: Create two variables of type `str` and add them together. What is the result?

In [ ]:
```

```

3 Help

There are a vast amount of Python resources available online (https://www.python.org/search/, https://docs.python.org/3/contents.html, https://docs.python.org/3/tutorial/). However, users can find local help by pressing `Ctrl+I` or `Cmd+I` in front of any object. You can also use the interactive help provided in the Spyder environment through the help menu or via the console, for example by entering `help(print)`. Finally, help will automatically appear after typing a left parenthesis next to an object.



In [23]:
```
help(print)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
```

```
    flush
      whether to forcibly flush the stream.
```

If we wish to get help regarding the use of the console, we can use `?`.

In [24]:
```
?
```

Exercise: Search for help on the `abs()` function and use this function on a number of your choice defined in a variable `a`.

In [ ]:
```

```

Exercise: Assign to a variable your first name and last name separated by a "." and calculate the length of the of the resulting string.

In [ ]:
```

```

4 Terminology

Python employs several concepts, unique to the language, that are important to define. You can find more detailed information in the glossary (https://docs.python.org/3/glossary.html#glossary).

4.1 Scripts

A script is simply a `.py` file where a set of instructions intended to be executed are stored. Running the script will create new variables or graphs in the workspace.

4.2 Functions

A function is a set of instruction lines organized according to a well-defined syntax (using the `def` keyword) and is reusable. It is used to perform a unique action corresponding to the function. The corresponding syntax is given in the following example, which simply adds two variables, `a` and `b`, and assigns the result to a variable `c`, which is returned using the `return` keyword as the function's output:

In [1]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

In [2]:
```
def sum_a_b(a,b):
    #This function performs the sum of a and b.
    c = a + b
    return c
```

In [3]:
```
sum_a_b(1,3)
```

Out [3]:
```
4
```

A function definition associates a function name with a function object in the current namespace, which can be seen below through the command `whos`.

In [4]:
```
whos
```

```
Variable   Type        Data/Info
-------------------------------
sum_a_b    function    <function sum_a_b at 0x00000266B13A7060>
```

Exercise: Given the following function

$$f(x) = ax^2 + bx$$

Create a function that takes `x` as an argument and returns $f(x)$ with $a = 3$ and $b = 2.4$

In [ ]:
```
```

4.3 Modules

A module is a Python file (`.py`) intended to be imported into scripts, the console, or other modules. It defines classes, functions, and variables meant to be used once imported. This allows for the reuse of functions written for one program in another without having to copy them. The `import` command can be used to import the entire module or parts of it. One can then access them using `module_name.object_name`. In the example below, the dummy_module.py contains the following instructions:

In [9]:
```
#content of the file: dummy_module.py
title = 'Tutorial'

students_number = 300

def number_group(n):
    print(n/3)
```

In [10]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

In [11]:
```
import dummy_module
```

In [12]:
```
whos
```

```
Variable       Type      Data/Info
---------------------------------
dummy_module   module    <module 'dummy_module' fr<...>\\Tuto\\dummy_module.py'>
```

In [13]:
```
print(dummy_module.title)
n = dummy_module.students_number
dummy_module.number_group(n)
```

```
Tutorial
100.0
```

Note that if we execute the contents of the file `dummy_module.py` directly in the console, we will create a function `number_group()`, as well as two variables: the string `title` and the integer `students_number`. Importing the same will only create the `dummy_module` module in the workspace.

4.4 Packages

They are a certain type of Python modules that can contain sub-modules or sub-packages. Technically, a package is a module that has a `__path__` attribute. Packages are a way to structure various modules using a "dotted" notation. For example, the module name A.B designates the sub-module B of package A. You can think of packages as directories in the file system and modules as files within those directories.

4.5 Libraries

A library can contain dozens, or even hundreds, of individual modules which can provide a wide range of functionalities. `Matplotlib` is a library that allows for creating graphs to visualize data (https://matplotlib.org). The Python standard library (https://docs.python.org/3/library/) contains hundreds of modules to perform common tasks, such as the `help()` function which invokes the help system.

4.6 Classes

A class can be seen as a blueprint containing attributes (or variables) and methods (or functions) that allow for the creation of instances of that class. Classes allow for the definition of new types of variables specific to the programmer. Instantiating a class creates an object of that class. The methods associated with a class have privileged access to the class's data, and the attributes act like global variables for the methods of the class. A class is defined using the `class` keyword followed by the class' name and a colon. For instance, we might choose to create a `student` class where attributes such as age or first name are defined, and a method is provided to calculate their probability of passing the year. Subsequently, we would use this class to create different student instances and assess their chances of success.

In [32]:
```
class student:
    def __init__(self, age, first_name, name, ects_obtained, bloc1, cycle):
        self.age = age
        self.prenom = first_name
        self.email = first_name + '.' + name + '@student.uliege.be'
        self.nom = name
        self.ects_obtained = ects_obtained
        self.bloc1 = bloc1
        self.cycle = cycle


    def proba_passing(self):
        pr = (self.age/self.ects_obtained)
        return pr


s932810 = student(20, 'John', 'Smith', 30, True, 'Bachelor')
```

```
s493304 = student(22, 'Jane', 'Doe', 30, False, 'Master')
```

In [33]:
```
print(s932810.email)
print(s493304.email)
```

```
John.Smith@student.uliege.be
Jane.Doe@student.uliege.be
```

In [34]:
```
print(s932810.proba_passing())
```

```
0.6666666666666666
```

In [35]:
```
print(s493304.proba_passing())
```

```
0.7333333333333333
```

The keyword `self` refers to the instance of the class on which the method will operate, and `__init__` is the class constructor that initializes the attributes of the class.

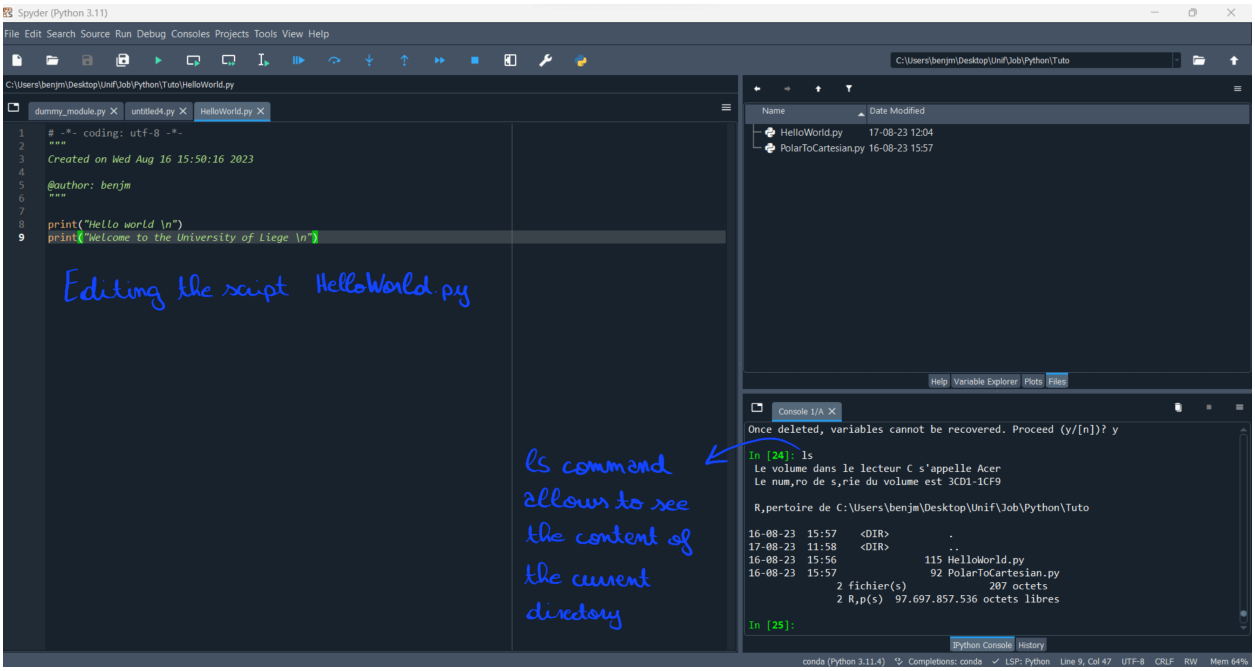
Exercise: Create your own `Professor` class with its attributes and methods, and find a use for it.

In [ ]:
```

```

5 Script Editing

The current directory is the directory where the user is working. The content of this directory is displayed in Spyder, and features for managing its content are provided. The content of the current directory can also be viewed in the command window using the `ls` command.

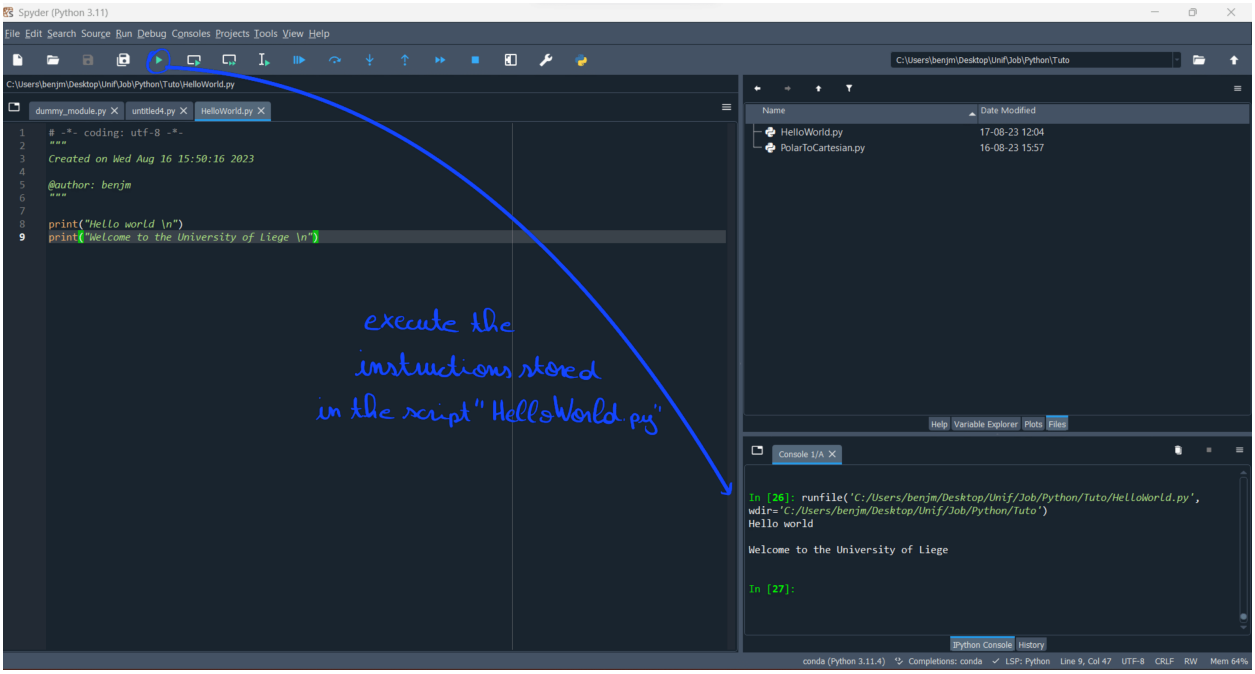


As defined above, a script is simply a file where a set of commands is saved, which will be executed in the console exactly as if the commands were entered directly there. Script files are saved in the current directory with the ".py" extension and can be launched from the editor in Spyder by clicking on run or by executing the command `%run` followed by the script file's name. Anything following the character `#` or between the characters `"""` is considered a comment and will therefore not be executed. A script can include simple instructions, module imports, and calls to functions, as long as those functions are defined in the workspace.

In [37]:
```
%run HelloWorld.py
```

```
Hello world

Welcome to the University of Liege
```



The same result is obtained when the commands from the script are entered directly into the console:

In [38]:
```
print("Hello world \n") #This is a comment
print("Welcome to the University of Liege \n")
```

```
Hello world

Welcome to the University of Liege
```

When executing a script, the entire script runs, and no function is called automatically, unlike other languages like `C` where the `main()` function is the first to be executed. This can be problematic when importing a script rather than running it directly. To avoid this, simply add `if __name__=='__main__':` within the script to specify that the subsequent code will only be exe-

cuted if the script is called directly and not imported. We illustrate this below with the ping.py and pingpong.py scripts.

In[172]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

In[173]:
```
#Script of pingpong.py
def ping():
    print("pong")
ping()
```

```
pong
```

In[174]:
```
#Script of ping.py

def ping():
    print("pong")

if __name__ ==  '__main__':
    ping()
```

```
pong
```

In[175]:
```
import pingpong
```

```
pong
```

In[176]:
```
import ping
```

In[177]:
```
whos
```

```
Variable   Type      Data/Info
------------------------------
ping       module    <module 'ping' from 'C:\\<...>\\Python\\Tuto\\ping.py'>
pingpong   module    <module 'pingpong' from '<...>thon\\Tuto\\pingpong.py'>
```

Exercise: Create a Python script in which a function `greet(name)` is defined that prints *Hello 'name'*, and use this function to display "Hello 'our name'" in the console.

In [ ]:
```

```

5.1 Indentation

Most programming languages have some form of a start-end structure, or opening and closing braces, or something similar to clearly define the boundaries of code within the loop, or in different parts of a structure. Generally, experienced programmers also indent their code so that it is easier for a reader to see what is inside a loop, especially if there are multiple nested loops. But in most languages, indentation is just a matter of style and the language's start-end structure determines how it is actually interpreted by the computer.

In Python, indentation is all-important. There is no start and end, just indentation. Anything that is meant to be at a level within a loop has to be indented at that level. Once the loop is finished, the indentation should go back to the previous level.

The number of spaces to indent at each level is a matter of style, but you must be consistent within a single code block. The standard is often 4 spaces, which is typically equivalent to the `Tab` key on the keyboard. However, this can vary depending on the configuration of your editor or development environment. Indentation makes the code readable. Do not hesitate to also add comments to increase the readability of your code using the `#` character or the `""" """` characters, which allow for multi-line comments.

In [14]:
```
"""
Created on Wed Jan 20 11:19:08 2021
This program calculates the cosine of i for i ranging from 0 to 3 with a step of
1 and displays the last value obtained. It requires no input parameters and
provides no output.

@author : Alan Turing
"""

# Import of libraries required for this script
import numpy


# Main body of the script
for i in [0, 1, 2, 3]:
    x = numpy.cos(i)

# Display of the value of x at the end of the loop
print('x equals', x)
```

```
x equals -0.9899924966004454
```

6 Libraries, functions, local and global variables

Upon starting Python, a number of basic functions are available, such as the `print()` function that allows you to display variables or strings, as well as the general language syntax. However, most of the functions needed for specific purposes, like simply calculating the cosine of an angle, are found in libraries that do not load by default in order to save startup time for Python. For instance, if we try to execute the `cos()` function, we encounter an error message :

In [45]:
```
cos(0)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[45], line 1
----> 1 cos(0)
```

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

The cosine function is available in the `NumPy` library. Libraries are imported into the workspace with the `import` command. Each library contains many functions. To use the `cos()` function from the `NumPy` library, we need to specify it. To do this, the syntax is as follows: `library.function()`.

In [46]:
```
import numpy
numpy.cos(0)
```

Out [46]: 1.0

In [47]:
```
type(numpy)
```

Out [47]: module

In [48]:
```
type(numpy.cos)
```

Out [48]: numpy.ufunc

We can also choose to import only the `cos()` function from the `NumPy` module and rename it, in this case, to `cosine`.

In [49]:
```
from numpy import cos as cosine
cosine(0)
```

Out [49]: 1.0

To define a function, one simply uses the `def` syntax. The commands entered below in the console define a function named "PolarToCartesian" which takes as input parameters the radius "rho" and the angle "theta" defined in degrees. These parameters follow the function name and are specified within parentheses. This function outputs the corresponding Cartesian coordinates "x" and "y" using the `return` command:

In [53]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

In [54]:
```
# Importing external libraries or functions
import numpy

# Function definition
def PolarToCartesian(rho,theta):
    """
    Parameters
    ----------
    rho : radius

    theta : angle in degrees
```

```
    Returns
    -------
    Cartesian coordinates of a point defined from its polar coordinates.

    """

    theta = theta * numpy.pi/180
    x = rho * numpy.cos(theta)
    y = rho * numpy.sin(theta)
    return x,y
```

In [57]:
```
whos
```

```
Variable           Type        Data/Info
----------------------------------------
PolarToCartesian   function    <function PolarToCartesian at 0x000001DEEF7109A0>
numpy              module      <module 'numpy' from
'C:\<...>ges\\numpy\\__init__.py'>
```

To call a function in the console once it is defined, you need to type its name followed by parentheses, inputting the function's arguments inside them (if the function requires any):

In [58]:
```
PolarToCartesian(1,0)
```

Out [58]:
```
(1.0, 0.0)
```

As can be seen above, the PolarToCartesian() function requires the use of the `NumPy` library to compute the cosine and sine of the given angle.

It is important to note that a function is a set of commands executed in a sub-environment which only exists during the function's execution. If we try to call the variable "x" outside of the PolarToCartesian() function, the console will return an error message. Calling `who` verifies that the function exists, but not the variables defined within that function. Therefore, variables defined within a function have a **local scope**.

In [59]:
```
x
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[59], line 1
----> 1 x

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

In [60]:
```
who
```

```
PolarToCartesian        numpy
```

A variable can have a **global scope** if it is declared outside of a function.

In [61]:
```
global_x = 'global variable'

def dummy_function():
    print('I am a ' + global_x)

dummy_function()
```

```
I am a global variable
```

If we want to be able to modify a global variable within a function, we need to declare it as `global`.

In [62]:
```
global_x = 'global variable'

def dummy_function():
    global global_x
    global_x = 'I am a ' + global_x

dummy_function()
print(global_x)
```

```
I am a global variable
```

Finally, Python offers the ability to declare a variable as **non-local** using the `nonlocal` keyword. This is used when nested functions are implemented in a script. In this way, the nested function has access to variables from the enclosing functions. Suppose we want to calculate the area of a disk and the circumference of a hole centered on this disk, where the hole's radius is half the size of the disk's radius.

In [64]:
```
import numpy

def circle(radius):
    pi = numpy.pi
    disk_area = pi * (radius) ** 2
    def circumference():
        nonlocal radius
        radius = radius/2
        circumference = 2 * pi * radius
        return circumference
    print(circumference())
    return disk_area
print(circle(3))
```

```
9.42477796076938
28.274333882308138
```

The scope of a variable in Python is thus the part of the code where it is visible. It can be local, non-local, global, and "built-in".

Except for very simple functions, you would not want to write functions directly into the console. Normally, you should create a `.py` file containing your function and import the resulting module into your interactive session or into your script. In the following example, we have copied all the commands from the PolarToCartesian function mentioned above into a file named "chgcoord.py". We import the module "chgcoor", which we rename to "coor", and then use the "PolarToCartesian" function defined within the module.

In [15]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

In [16]:
```
import chgcoor as coor
```

In [17]:
```
whos
```

```
Variable   Type      Data/Info
------------------------------
coor       module    <module 'chgcoor' from 'C<...>ython\\Tuto\\chgcoor.py'>
```

In [18]:
```
coor.PolarToCartesian(1,0)
```

Out [18]:
```
(1.0, 0.0)
```

6.1 List of Useful Libraries

Below we list a series of useful libraries and functions. Do not hesitate to read the corresponding help for the various functions provided and to test them out for yourself.

`matplotlib`

Useful module: pyplot

Examples of Useful Functions: plot() legend() xlabel() ylabel() show() xlim() grid()

`numpy`

Examples of Useful Functions: abs() sum() zeros() asarray() arange() min() max() where() linspace() shape() loadtxt() floor() ceil() log() cos() sqrt()

`scipy`

Examples of Useful Functions: integrate.solve_ivp() interpolate.splrep() interpolate.splev()

Exercise: Given the quadratic equation

$$ax^2 + bx + c = 0,$$

create a function that takes the coefficients a, b, c as arguments and returns the two roots.

In [ ]:
```

```

7 Other types of objects: list, tuple, and array

We will use the `numpy` library to perform mathematical operations on "arrays". In addition to these arrays, Python has other data types that can store sequences of items: "lists" and "tuples". However, these are not used for mathematical operations.

7.1 List and tuple

A list is defined by `[ ]` while a tuple is defined by `( )`. You can modify the values in a list but not in a tuple.

In [70]:
```
a_list=[2, 4, 6, 8, 10]
print(a_list)
type(a_list)
```

```
[2, 4, 6, 8, 10]
```

Out [70]: `list`

In [71]:
```
a_tuple=(2, 4, 6, 8, 10)
print(a_tuple)
type(a_tuple)
```

```
(2, 4, 6, 8, 10)
```

Out [71]: `tuple`

If we attempt to perform an operation on these two objects, we see that what might be considered as a mathematical operation actually creates a new list containing `a_list` twice, while it is not possible to modify `a_tuple`.

In [72]:
```
a_list * 2
```

Out [72]: `[2, 4, 6, 8, 10, 2, 4, 6, 8, 10]`

In [73]:
```
a_tuple + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[73], line 1
----> 1 a_tuple + 2

TypeError: can only concatenate tuple (not "int") to tuple
```

Lists and tuples can contain sequences of various types, not just numbers. In the example below, we create a list containing a string, numbers, and a function.

In [74]:
```
a_varied_list = ['list', 1, 20.45, print]
print(a_varied_list)
```

```
['list', 1, 20.45, <built-in function print>]
```

7.2 1-dimensional Arrays

To perform mathematical operations on objects containing sequences of values, we will use arrays. For this, we will need the libraries `NumPy` and `matplotlib`.

In [75]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

Then, we can use various functions to create a 1-dimensional array, for example, ranging from 1 to 5:

In [76]:
```
x = np.array([1, 2, 3, 4, 5]) # We have manually inserted the individual
                              # elements of the array using a list
print('x:', x, type(x))

y = np.arange(1,6,1) #Starts from 1, stops before 6 with a step of 1
print('y:', y, type(y))

z = np.linspace(1,5,5) #Starts from 1 up to 5 with 5 points spaced uniformly
print('z:', z, type(z))
```

```
x: [1 2 3 4 5] <class 'numpy.ndarray'>
y: [1 2 3 4 5] <class 'numpy.ndarray'>
z: [1. 2. 3. 4. 5.] <class 'numpy.ndarray'>
```

By using the `len` (built-in) function, `shape` (useful for arrays with more than 1 dimension), and `ndim` (methods of `ndarray`), we can determine the lengths, shapes and dimensions of the arrays:

In [77]:
```
print(len(x))
```

```
5
```

In [78]:
```
x.shape
```

Out [78]: (5,)

In [79]:
```
x.ndim
```

Out [79]: 1

To access the elements of arrays, we use their linear indices starting from 0. Thus, the first element of a 1-dimensional array has the index `0`. The last element of a 1-dimensional array can be accessed using the index `-1`. Indices are specified using square brackets `[index]`. The values of the elements can then be modified.

In [80]:
```
print('The value of the first element of x is', x[0])

x[0] = 10
```

```
print('The first element of x is now', x[0])
```

```
The value of the first element of x is 1
The first element of x is now 10
```

In [81]:
```
x[-1]
```

Out [81]:
```
5
```

We can also access multiple elements by using : and specifying the start, end, and step.

In [82]:
```
x[0:4:2] #x[start:before_end:step]
```

Out [82]:
```
array([10,  3])
```

In [83]:
```
x[:]
```

Out [83]:
```
array([10,  2,  3,  4,  5])
```

It is also possible to go through arrays in reverse.

In [84]:
```
x[len(x)::-1]
```

Out [84]:
```
array([ 5,  4,  3,  2, 10])
```

We can also search for specific elements in an `array` using the `where` function, as illustrated below:

In [85]:
```
a = np.array([-1, 0, 1.3, 4, -10, 0.4, 3, 9, 1])
ind_a_neg = np.where(a < 0)
print('The indices corresponding to negative values are:', ind_a_neg)
print('The negative values of a are:', a[ind_a_neg])
```

```
The indices corresponding to negative values are: (array([0, 4], dtype=int64),)
The negative values of a are: [ -1. -10.]
```

Slices in Python allow for segmenting objects containing sequences of values from an object `a` (nparray, list, tuple, string) by selecting only a portion of them. The syntax for slices is `a[i:j]` or `s[i:j:k]`, where the indices i (start), j (end), and k (increment) can be omitted.

In [86]:
```
a = 'abcdefghijklmnopqrstuvwxyz'
type(a)
```

Out [86]:
```
str
```

In [87]:
```
a[0:2]
```

Out [87]:
```
'ab'
```

In [88]:
```
a[0:]
```

Out [88]: 'abcdefghijklmnopqrstuvwxyz'

In [89]:
```
a[0:26:2]
```

Out [89]: 'acegikmoqsuwy'

In [90]:
```
a[:12]
```

Out [90]: 'abcdefghijkl'

In [91]:
```
a[:]
```

Out [91]: 'abcdefghijklmnopqrstuvwxyz'

In [92]:
```
a[-1]
```

Out [92]: 'z'

In [93]:
```
a_tuple=(1, 2, 3, 'quatre')
```

In [94]:
```
a_tuple[2:4]
```

Out [94]: (3, 'quatre')

In [95]:
```
a_list=[1, 3, 4, 4]
a_list[0:3]
```

Out [95]: [1, 3, 4]

In [96]:
```
a_array=np.array(a_list)
a_array[0:3]
```

Out [96]: array([1, 3, 4])

7.2.1 Operations on a 1-dimensional array

When performing operations on `NumPy arrays`, the execution speed can be greatly improved by avoiding the use of loops to traverse the arrays and by using the various `NumPy` functions that can be applied directly to the arrays.

In [97]:
```
x = np.arange(1,6,1)
y = np.zeros(x.shape)  # initializing y to the same size as x
for i in range(len(x)):
    y[i] = np.sin(x[i])
print(y)
```

```
[ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427]
```

This can be simply written as:

In [98]:
```
y = np.sin(x)
print(y)
```

```
[ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427]
```

Vectorized operations can be applied to 1-dimensional arrays of type `array` using the appropriate functions from `NumPy`, while basic operations apply element by element. For example, the multiplication of x by y using the `*` operator yields:

In [99]:
```
x = np.arange(1,6,1)
y = np.arange(0,5,1)
print('x:', x)
print('y:', y)
x * y
```

```
x: [1 2 3 4 5]
y: [0 1 2 3 4]
```

Out [99]:
```
array([ 0,  2,  6, 12, 20])
```

To compute the dot product of vectors x and y, we can use the `dot` function from `NumPy`.

In [100]:
```
np.dot(x,y)
```

Out [100]:
```
40
```

Exercise: Create a one-dimensional array (row vector) `x` with 5 consecutive numbers between 2. and 3., spaced by equal intervals.

In [ ]:
```

```

Exercise: Add 1 to the second element of `x`.

In [ ]:
```

```

Exercise: Create a second row array `y` of the same dimension as `x`, but with elements being consecutive even numbers starting at 4.

In [ ]:
```

```

7.3 2-Dimensional Arrays

Arrays can be of arbitrary dimensions, but 2-dimensional arrays, for which matrix operations can be defined, are frequently used. To create these arrays, we can reuse the `array` function from `NumPy`.

In [101]:
```
A = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]])
print(A)
print('A has', np.ndim(A), 'dimensions')
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
```

```
 [11 12 13 14 15]]
A has 2 dimensions
```

In[102]:
```
print('The size of A is:', A.shape)
```

```
The size of A is: (3, 5)
```

The `flatten` and `reshape` functions from `NumPy` allow for modifying the dimensions of arrays. `flatten` transforms a multi-dimensional array into a one-dimensional array, while `reshape` lets you specify the dimensions of the array you wish to obtain. `flatten` does not require any input parameters, whereas `reshape` requires that the dimensions of the two arrays be compatible.

In[103]:
```
B = A.flatten()
print(A.flatten())
print('A.flatten() has', np.ndim(B), 'dimension(s)')
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
A.flatten() has 1 dimension(s)
```

In[104]:
```
B.reshape(3,5)
```

Out[104]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

In[105]:
```
B.reshape(3,6)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[105], line 1
----> 1 B.reshape(3,6)

ValueError: cannot reshape array of size 15 into shape (3,6)
```

The index of a 2-dimensional array is specified by two values: the first corresponds to the rows, and the second to the columns of the array. These are provided using a tuple (thus defined in parentheses). In the example below, we create a 2-dimensional array filled with 0s, and we modify it by changing one element, then part of a row, and finally the last element.

In[106]:
```
B = np.zeros((3,5))
print('Initialization of B:')
print(B)
print('Modification of B:')
B[0,0] = 5
B[1,2:] = 10
B[-1,-1] = 500
print(B)
```

```
Initialization of B:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
Modification of B:
[[  5.   0.   0.   0.   0.]
 [  0.   0.  10.  10.  10.]
 [  0.   0.   0.   0. 500.]]
```

As before, the basic operations are performed element by element:

In[107]:
```
print(A * B)
```

```
[[5.0e+00 0.0e+00 0.0e+00 0.0e+00 0.0e+00]
 [0.0e+00 0.0e+00 8.0e+01 9.0e+01 1.0e+02]
 [0.0e+00 0.0e+00 0.0e+00 0.0e+00 7.5e+03]]
```

The `array` object `A` from `NumPy`, has some automatically defined methods, such as the transpose of `A`, denoted as `A.T`. Matrix operations can be performed using the functions of `NumPy`.

In[108]:
```
print(np.dot(A.T,B))
```

```
[[5.00e+00 0.00e+00 6.00e+01 6.00e+01 5.56e+03]
 [1.00e+01 0.00e+00 7.00e+01 7.00e+01 6.07e+03]
 [1.50e+01 0.00e+00 8.00e+01 8.00e+01 6.58e+03]
 [2.00e+01 0.00e+00 9.00e+01 9.00e+01 7.09e+03]
 [2.50e+01 0.00e+00 1.00e+02 1.00e+02 7.60e+03]]
```

Exercise: Create a 2-dimensional `ndarray` called `A` where the first row is equal to `x`, the second row is filled with 1s, and the third row is equal to `y`. `x and y` are defined in the previous exercise.

In [ ]:
```
```

Exercise: Consider two matrices:

$$A = \begin{bmatrix} 4 & -5 \\ \sqrt{3} & \pi/4 \end{bmatrix}, \; B = \begin{bmatrix} 2 & 3+i \\ -72/3 & 0.2 \end{bmatrix}$$

where i is the imaginary unit. Calculate the following elements:

$$A + B, \; AB, \; A^2, \; A^T, \; B^{-1}, \; B^T A^T, \; A^2 + B^2 - AB$$

In [ ]:
```
```

8 Solving Systems of Linear Equations

To solve a linear system Ax = b, for example,

$$\begin{pmatrix} 5 & 6 & 10 \\ -3 & 0 & 14 \\ 0 & -7 & 21 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 10 \\ 0 \end{pmatrix}$$

We can use the `solve` method, which is part of the `linalg` sub-library of `NumPy`. The solving method takes a two-dimensional array (the matrix A) and a one-dimensional array (the right-hand side, b) as input, and it returns the solution.

In[109]:
```
A = np.array([[5, 6, 10],[-3, 0, 14], [0, -7, 21]])
b = np.array([4, 10, 0])
solution = np.linalg.solve(A, b)
print(solution)
```

```
[-1.45454545  1.20779221  0.4025974 ]
```

One can simply verify that the provided solution is correct by multiplying matrix A by the solution.

In[110]:
```
np.dot(A,solution)
```

Out[110]:
```
array([ 4., 10.,  0.])
```

Exercise: Solve the following system of linear equations:

$$5.4x + 2y = 0 - x + 4y - 25z = 33x + 7z = 2$$

In [ ]:
```

```

9 Control Structures: `for`, `if/elif/else`,`while`, `break`

Python comes with built-in control structures. They allow for loops, checking conditions before executing certain lines of code. We will need `numpy` and `matplotlib` here, so we start by importing them.

9.1 The `for` Loop

The syntax is as follows and allows executing a block of commands repeatedly:

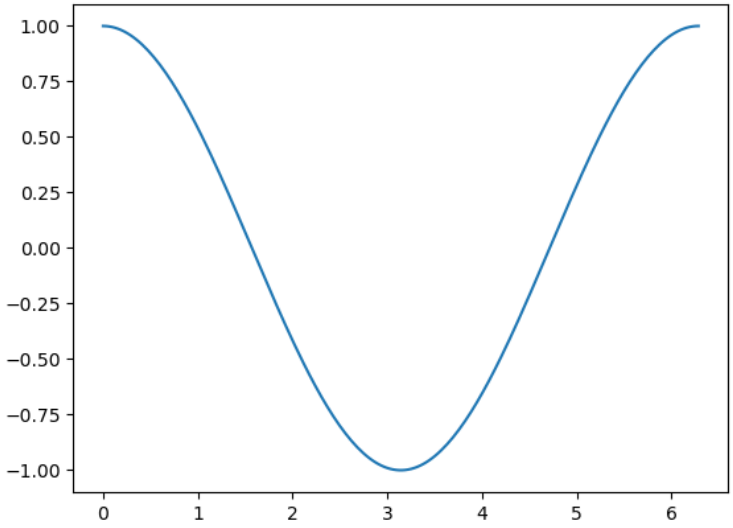
In[111]:
```
for i in [0, 1, 2, 3, 4]:  # The line ends with ":"
    print('The value of i is:', i)  # The block of commands is indented
print('We have exited the loop')
```

```
The value of i is: 0
The value of i is: 1
The value of i is: 2
The value of i is: 3
The value of i is: 4
We have exited the loop
```

It should be noted here that at the end of the `for`, it is necessary to introduce `:` and indent the block of commands to be executed within the loop. The end of the indentation marks the end of the loop.

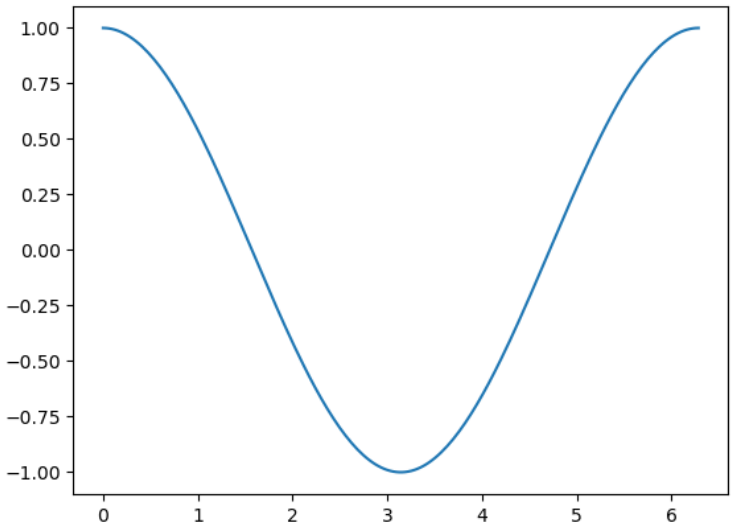
We can use `for` loops to assign values to elements of an array.

In[112]:
```
x = np.linspace(0, 2*np.pi, 200)
y = np.zeros_like(x)
for i in range(len(x)):
    y[i] = np.cos(x[i])
plt.plot(x,y);
```



Note that we should have taken advantage of the vectorization offered by `NumPy` and avoided writing a `for` loop:

In[113]:
```
plt.plot(x,np.cos(x));
```



9.2 `if/elif/else`

The `if` condition allows executing a block of commands as long as the result of the evaluation of the `if` is `True`.

In[114]:
```
a = 2
if a == 2:
    print('a is indeed equal to', a)
```

```
a is indeed equal to 2
```

The `if` condition can be followed by an `else` which will be evaluated if the evaluation of the `if` is `False`.

In[115]:
```
a = 3
if a == 2:
    print('a is indeed equal to', a)
else:
    print('a is not equal to 2')
```

```
a is not equal to 2
```

Finally, we can use the `elif` command to add conditions before using the `else`.

In[116]:
```
a = 3
if a == 2:
    print('a is equal to', a)
elif a == 3:
    print('a is equal to', a)
else:
    print('a is neither equal to 2 nor 3')
```

```
a is equal to 3
```

9.3 The `while` Loop / `break` / `continue`

The `while` loop allows executing a block of commands until a condition is met or when a `break` command is encountered. The syntax is as follows:

In[117]:
```
a = 0
while a < 6:
    a += 1
    print(a)
```

```
1
2
3
4
5
6
```

It is important to be cautious not to create an infinite loop and to use the `break` command after a certain number of iterations in the loop.

In[118]:
```
a = 0
while a < 60000:
    a += 1
    print(a)
    if a > 15:
        print('Early loop exit')
        break
print('We have exited the loop')
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Early loop exit
We have exited the loop
```

The `continue` statement is used to skip the remaining code within a loop for the current iteration only.

In[119]:
```
a = 0
while a < 5:
    a += 1
    if a == 3:
        continue
    print(a)
```

```
1
2
4
5
```

Exercise: The exponential function has the series expansion:

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Write a Python function that takes the input variable `n` and calculates this series up to the first `n+1` terms.

In [ ]:
```
```

10 Polynomials

Polynomials can be designed and manipulated using the `numpy.polynomial` class, which provides various options (power, Chebyshev, Legendre, etc.). For example, the polynomial

$$x^4 - 12x^3 + 5x$$

can be represented by its coefficients listed in the `coef` list below.

In [25]:
```
coef = [0, 5, 0, -12, 1]
```

In [26]:
```
from numpy.polynomial import Polynomial as poly
p = poly(coef)
print(p)
```

```
0.0 + 5.0 x + 0.0 x**2 - 12.0 x**3 + 1.0 x**4
```

In [27]:
```
type(p)
```

Out [27]:
```
numpy.polynomial.polynomial.Polynomial
```

We can then perform a variety of operations such as calculating the derivative of the polynomial, evaluating it at a point (for example, at `x = 5`), or determining its roots.

In [28]:
```
dp = p.deriv(1)
print(dp)
```

```
5.0 + 0.0 x - 36.0 x**2 + 4.0 x**3
```

In [29]:
```
p(5)
```

Out [29]:
```
-850.0
```

In [30]:
```
p.roots()
```

Out [30]:
```
array([-0.62921183,  0.        ,  0.66413705, 11.96507478])
```

If a set of points is given to us but we do not know the associated polynomial, we can fit a polynomial of a certain degree to these points using the least squares method with the `polyfit` function.

In [31]:
```
from numpy.polynomial.polynomial import polyfit
coef2 = polyfit([0, 1, 4, 5],[-1, 2, 1, 4],3)
```

In [32]:
```
p2 = poly(coef2)
print(p2)
```

```
-1.0 + 5.16666667 x - 2.5 x**2 + 0.33333333 x**3
```

11 Graphics or Plots

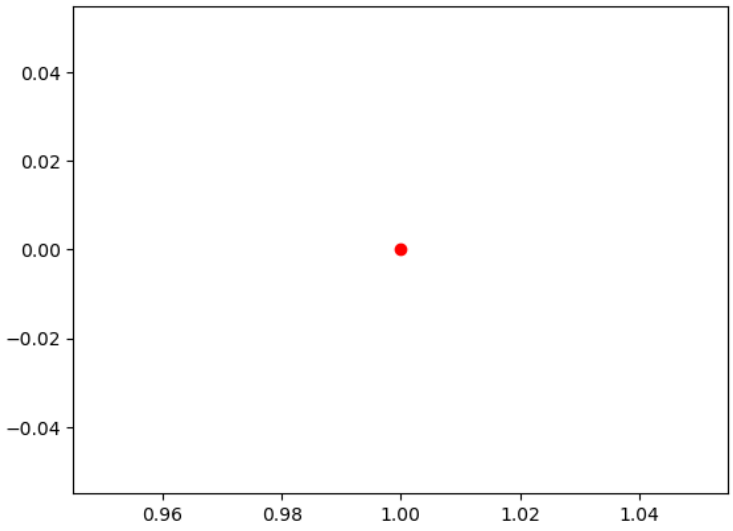
11.1 Basics

To create graphs in Python, we will use the `matplotlib` library and the sub-library `pyplot`. The full set of available graphs is described here: https://matplotlib.org/gallery.html (The command `%matplotlib inline` is used in this tutorial to prevent the graphs from appearing on a separate page but is not necessary when using Spyder.)

In [36]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

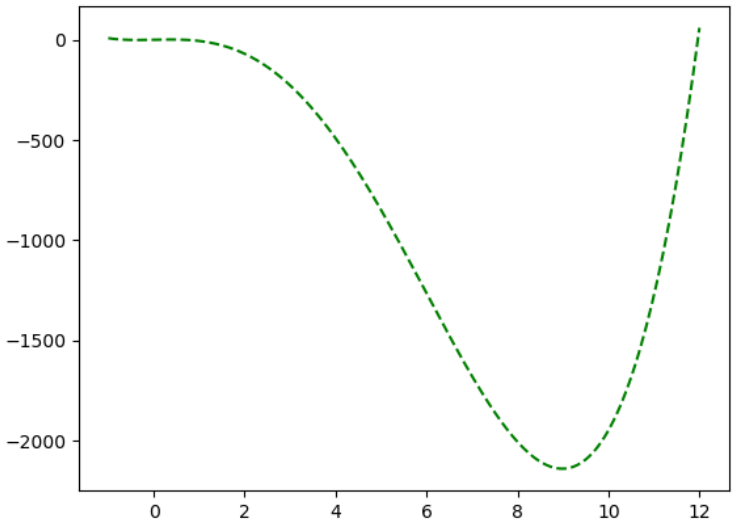
We can now utilize the plot functions using the command `plt.function`, for example the function `plot(x, y, string)` where the string defines the type of marker or line style used. In the following example, "o" stands for a circle and "r" for the colour red.

In [37]:
```
plt.plot(1,0,'or');
```



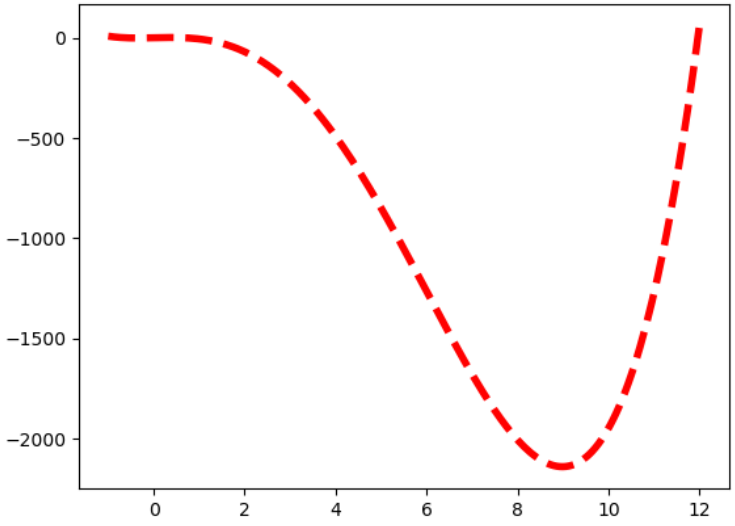
We can thus visualize the polynomial generated above using the arrays `x` and `y`.

In [38]:
```
x = np.linspace(-1, 12, 200)
plt.plot(x,p(x),'g--');
```



The `plot` function can accept numerous optional keyword arguments. These are specified in the `plot` function as arguments with the syntax `keyword=value`. For example, to plot the polynomial in red with a line thickness of 4, one would use:

In [39]:
```
plt.plot(x,p(x),'r--',linewidth = 4);
```



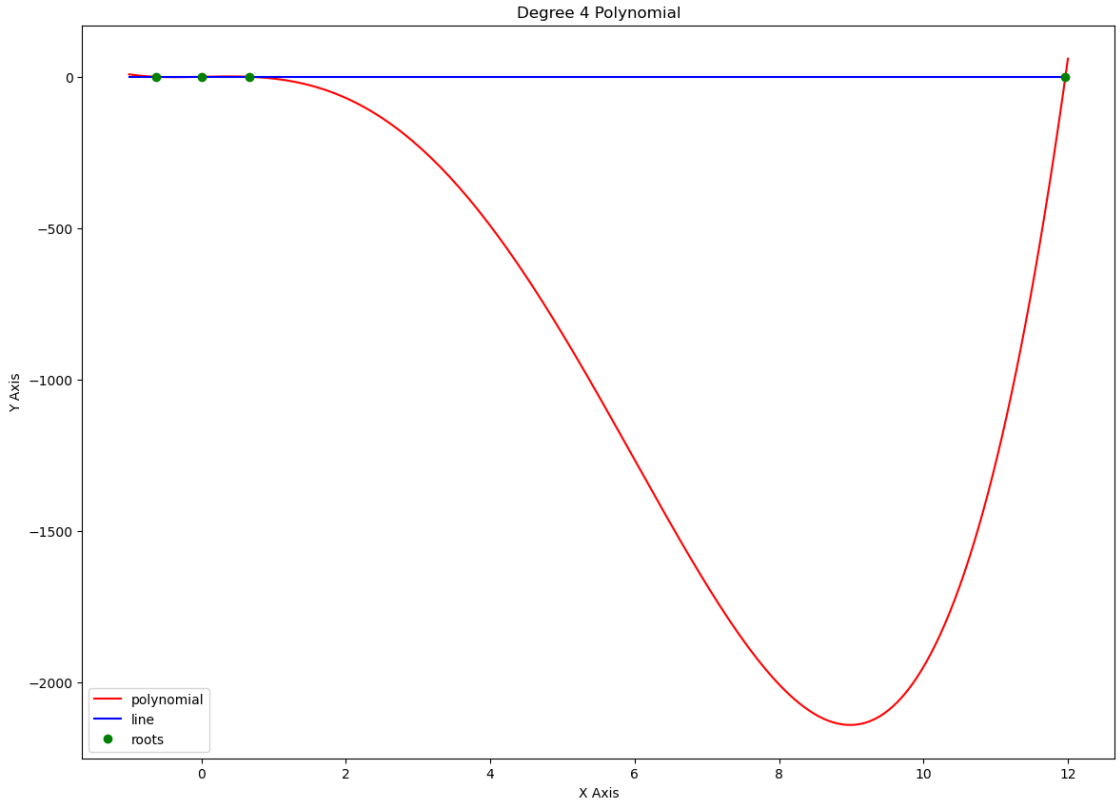
11.2 Display multiple curves in one graph, create a new figure

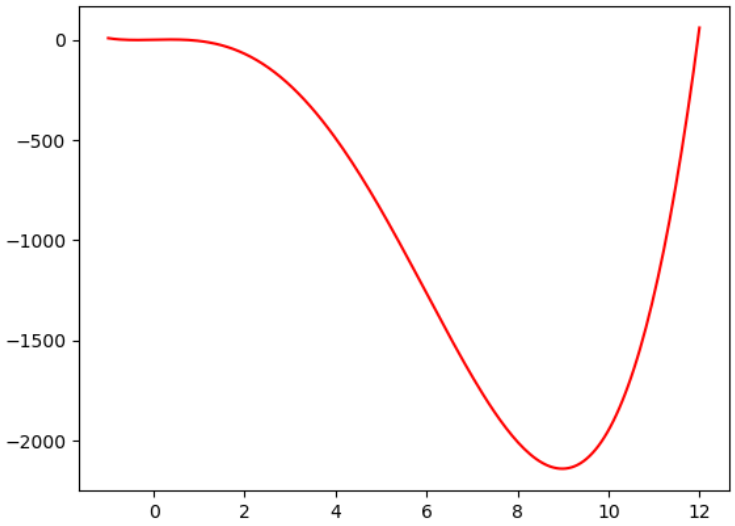
To create a new figure, you need to use the `figure` function from `matplotlib.pyplot`. You can then specify the size of the figure. The subsequent commands using the `plot` function will then use this figure, for instance, to overlay curves. You can also add a legend, a title, and label the axes.

In[137]:
```
plt.figure(figsize=(14, 10))  # Create a new figure
plt.plot(x, p(x), 'r');
plt.plot(x, np.zeros_like(x), 'b');
plt.plot(p.roots(), np.zeros_like(p.roots()), 'og')
plt.legend(['polynomial', 'line', 'roots'], loc='best')  # Add a legend

plt.title('Degree 4 Polynomial')  # Add a title
plt.ylabel('Y Axis')  # Labeling the y-axis
plt.xlabel('X Axis')  # Labeling the x-axis

plt.figure()  # Create a new figure
plt.plot(x, p(x), 'r');
```





Exercise: Plot the graph of the function

$$y(x) = e^{-0.8x}\sin\omega x$$

for $\omega = 10$ rad/s and $x \in [0\ 10]$ s.

In [ ]:

Exercise: Given the function:

$$y(x) = 10 + 5e^{-x}\cos(\omega x + 0.5)$$

Write a script that plots the graph for three distinct values of ω: 1 rad/s, 3 rad/s and 10 rad/s over the range $x \in [0\ \ 5]$ s. The three curves should appear in green, with a solid line for $\omega = 1$ rad/s, a dashed line for $\omega = 3$ rad/s, and a dotted line for $\omega = 10$ rad/s.

In [ ]:

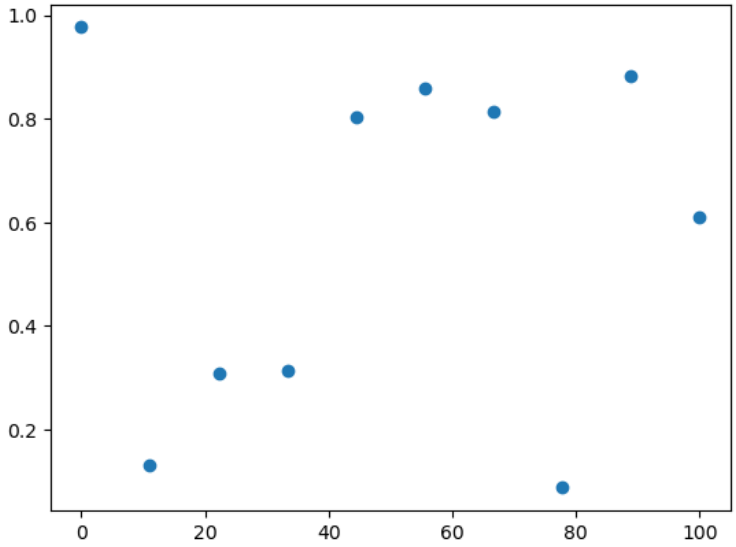
12 Interpolation using splines

Python offers the functionality to perform spline interpolations through the `SciPy` library and its submodule `interpolate`. We will use, for instance, the `CubicSpline` function.

In[138]:
```
import numpy as np
import matplotlib
from scipy.interpolate import CubicSpline
%matplotlib inline
import matplotlib.pyplot as plt
```

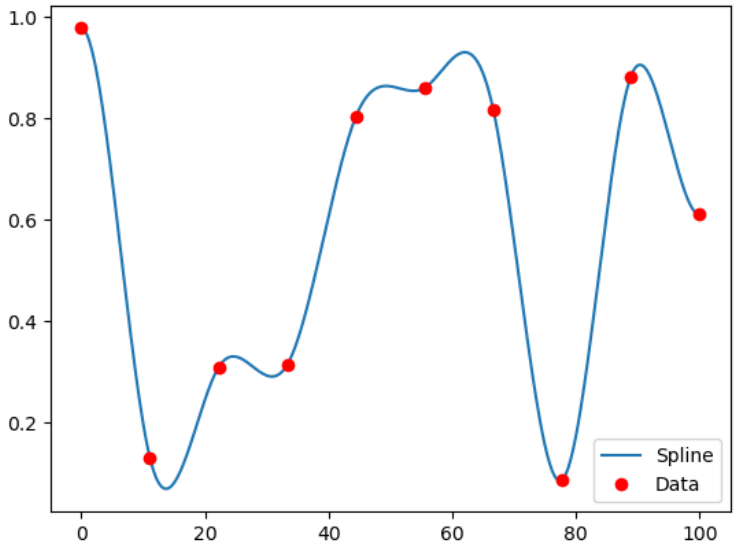
First, let's consider a set of points x and y between which we wish to interpolate.

In[139]:
```
x = np.linspace(0,100,10)
y = np.random.rand(10)
plt.plot(x,y,'o');
```



In[140]:
```
x_cs = np.linspace(0,100,500)
cs = CubicSpline(x,y,bc_type='clamped') # the bc_type parameter allows setting
                                        # certain boundary conditions
plt.plot(x_cs,cs(x_cs),label='Spline')
plt.plot(x,y,'ro', label='Data')
plt.legend(loc='best')
```

Out[140]: <matplotlib.legend.Legend at 0x1def766d750>



13 Inputs and Outputs

Instead of generating points randomly as in the cubic spline example, we might need to import them from an external source. For instance, these points could be contained in the following .txt file named `data.txt`:

In [ ]:
```
0. -1.
1. 2.
2. 0.
3. 0.1
4. -0.1
5. -2.
6. -8.
7. -9.
```

The `loadtxt` function from `NumPy` allows us to open this file, read its contents, and store them in a new variable.

In[141]:
```
import numpy as np
a = np.loadtxt('data.txt')
print(a)
type(a)
```

```
[[ 0.  -1. ]
 [ 1.   2. ]
 [ 3.   0.1]
 [ 4.  -0.1]
 [ 5.  -2. ]
 [ 6.  -8. ]
 [ 7.  -9. ]]
```

Out[141]:
```
numpy.ndarray
```

We can now perform mathematical operations on the created array (here we normalize the second column) and then save it as a `.txt` format using the `savetxt` function from `NumPy`.

In[142]:
```
a[:,1]=a[:,1]/np.max(a[:,1])
np.savetxt('normalised_data.txt',a)
print(a)
```

```
[[ 0.   -0.5 ]
 [ 1.    1.  ]
 [ 3.    0.05]
 [ 4.   -0.05]
 [ 5.   -1.  ]
 [ 6.   -4.  ]
 [ 7.   -4.5 ]]
```

Other options are available through basic functions like `open`, which is used to read, write, and modify a file, or `input()`, which allows for keyboard input. Numerous modules also exist for

reading various types of files, such as `xlrd`, which is used to read files from Microsoft Excel (`.xlx`, `.xls`).

In[143]:
```
first_name = input('What is your first name? :')
```

```
What is your first name? :Albert
```

In[144]:
```
print(first_name)
```

```
Albert
```

In[145]:
```
with open('data.txt') as f_data:
    print(f_data.read())
```

```
0. -1.
1. 2.
3. 0.1
4. -0.1
5. -2.
6. -8.
7. -9.
```

Exercise: Plot the graph of the cubic spline interpolation and the linear interpolation of the data points from the 'normalised_data.txt' file above.

In [ ]:
```

```

14 Solving differential equations

Consider the differential equation of the Van der Pol oscillator:

$$\frac{dx_1}{dt} = x_2 \tag{1}$$

$$\frac{dx_2}{dt} = \epsilon\omega(1 - x_1^2)x_2 - \omega^2 x_1 \tag{2}$$

With the constants ϵ and ω equal to 0.1 and 1, respectively. In Python, this differential equation can be represented by a function defined as follows:

In[146]:
```
def odefunction(t, y, const):
    # Ordinary differential equation system defined by:
    #   t: the time
    #   y: system variables
    # The function returns dy, an array containing the derivatives

    # Necessary import
    import numpy as np

    # Main part
    # Constant definitions
```

```
    epsilon = const[0]
    omega = const[1]

    dy = np.zeros(2)
    dy[0] = y[1]
    dy[1] = epsilon * omega * (1 - y[0]**2) * y[1] - (omega**2) * y[0]

    return dy
```

These lines of code are saved in a file named `OdeFun.py`. The numerical solution requires the `SciPy` library and the `solve_ivp` function from the `scipy.integrate` sub-library. Here, the output from `odefunction` provides a solution of type `list`, but we could also output a type `numpy.ndarray`.

In[147]:
```
%reset
from scipy.integrate import solve_ivp as ode45 # Defining the solver (ode45)
from matplotlib import pyplot
import numpy
import OdeFun
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

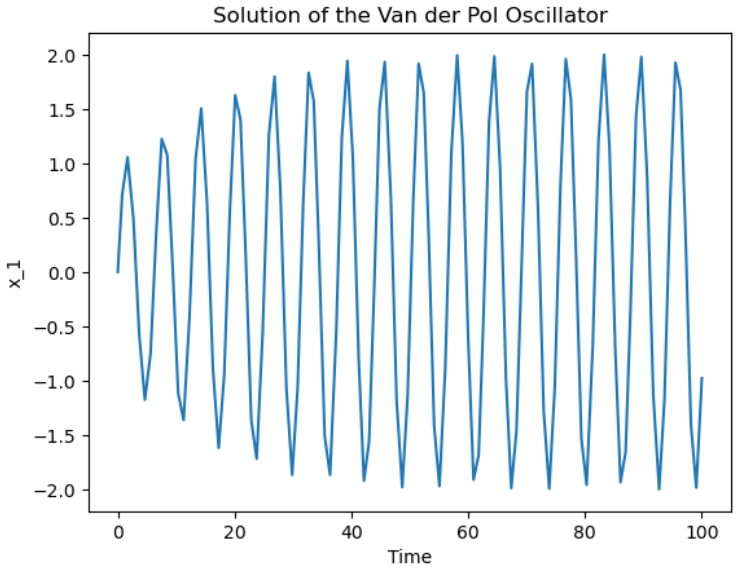
In[148]:
```
whos
```

```
Variable   Type        Data/Info
--------------------------------
OdeFun     module      <module 'OdeFun' from 'C:<...>Python\\Tuto\\OdeFun.py'>
numpy      module      <module 'numpy' from 'C:\<...>ges\\numpy\\__init__.py'>
ode45      function    <function solve_ivp at 0x000001DEF4C6D760>
pyplot     module      <module 'matplotlib.pyplo<...>\\matplotlib\\pyplot.py'>
```

The function `scipy.integrate.solve_ivp`, which is renamed here as `ode45`, uses by default an explicit Runge-Kutta method to solve the equation, and the integration parameters can be adjusted with options. The following code solves the differential equation between 0 and 100 and displays the result on the screen:

In[149]:
```
c = [0.1, 1]
solution = ode45(lambda t, y: OdeFun.odefunction(t, y, c), [0, 100], [0, 1])
pyplot.plot(solution.t, solution.y[0, :])
pyplot.title('Solution of the Van der Pol Oscillator')  # Adding a title
pyplot.ylabel('x_1')  # Labeling the y-axis
pyplot.xlabel('Time')  # Labeling the x-axis
```

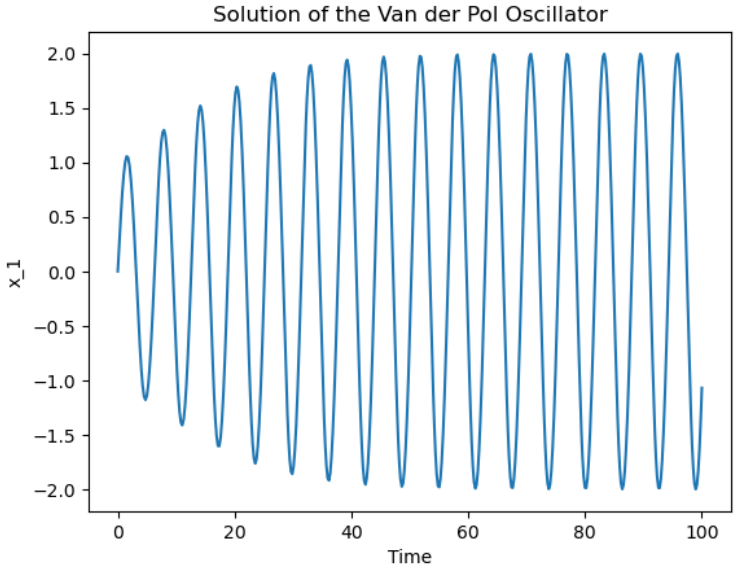
Out[149]:
```
Text(0.5, 0, 'Time')
```



The `lambda` function serves to replace the call to `OdeFun.odefunction`, which normally takes three arguments (t, y, c) as input, with a call to a `lambda` function with two arguments (t, y) that is accepted by the solver `scipy.integrate.solve_ivp`.

The solver `scipy.integrate.solve_ivp` provides a significant number of options. These options, for instance, allow defining values for relative and absolute tolerance.

In[150]:
```
solution = ode45(lambda t, y: OdeFun.odefunction(t,y,c), [0,100], [0, 1],␣
 ↪rtol=1e-8)
pyplot.plot(solution.t, solution.y[0,:]);
pyplot.title('Solution of the Van der Pol Oscillator'); # Adding a title
pyplot.ylabel('x_1'); # Labeling the y-axis
pyplot.xlabel('Time'); # Labeling the x-axis
```

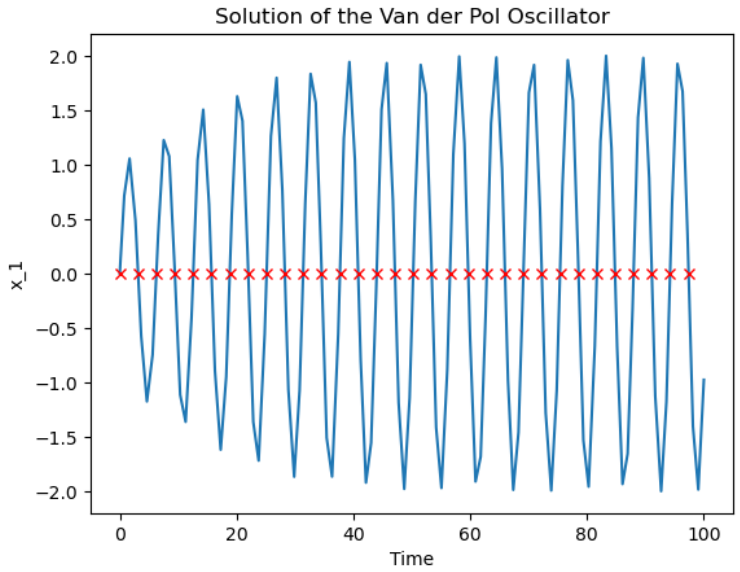


It is often useful to determine when the solution to the system of differential equations reaches a particular value (such as 0, for example). This can be achieved by defining specific options. This requires creating a function that defines the type of event to determine, and then we recalculate the solution. The events are stored in `solution.t_events` and `solution.y_events`.

For more information, please refer to the documentation for the `scipy.integrate.solve_ivp` function.

In[151]:
```
def event(t,y):
    return y[0]
```

In[152]:
```
solution = ode45(lambda t, y: OdeFun.odefunction(t,y,c), [0,100], [0, 1],␣
↪events=event)
pyplot.plot(solution.t, solution.y[0,:]);
pyplot.plot(solution.t_events[0], solution.y_events[0][:,0], 'rx');
pyplot.title('Solution of the Van der Pol Oscillator'); # Adding a title
pyplot.ylabel('x_1'); # Labeling the y-axis
pyplot.xlabel('Time'); # Labeling the x-axis
```



Exercise: For the above Van der Pol oscillator, find all the points where the value of x_1 is 1 and plot them on a graph illustrating this.

In [ ]:
```
```

Exercise: Solve the following differential equation:

$$\frac{dy_1}{dt} = cos(y_2) * y_3$$
$$\frac{dy_2}{dt} = -y_1/y_3$$
$$\frac{dy_3}{dt} = -0.8\, y_1 y_2$$

with the initial conditions $y_1(0) = 0$, $y_2(0) = 1$ and $y_3(0) = 1$ over the time range from 0 to 100. Plot the evolution of the three variables on the same graph. Find the zeros of the variable y_2 if any exist.

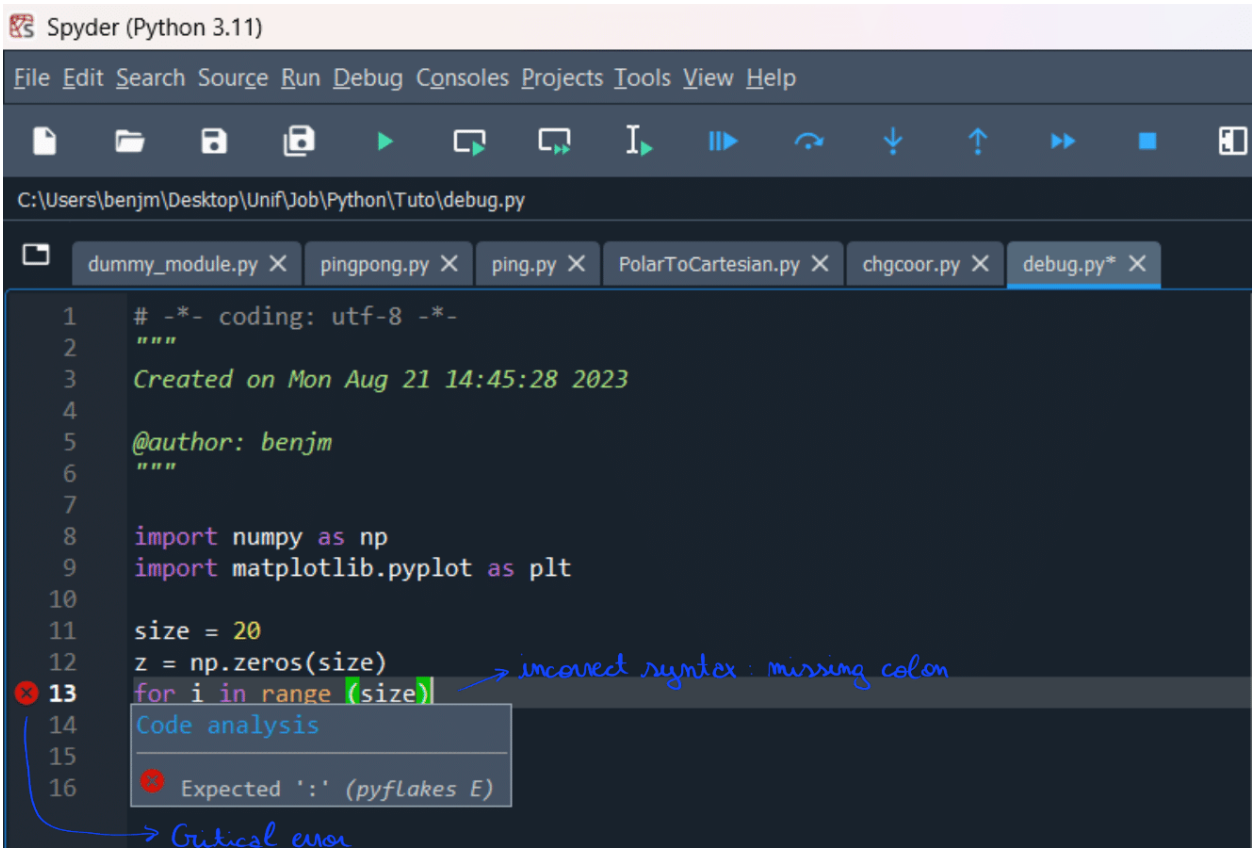
In [ ]:

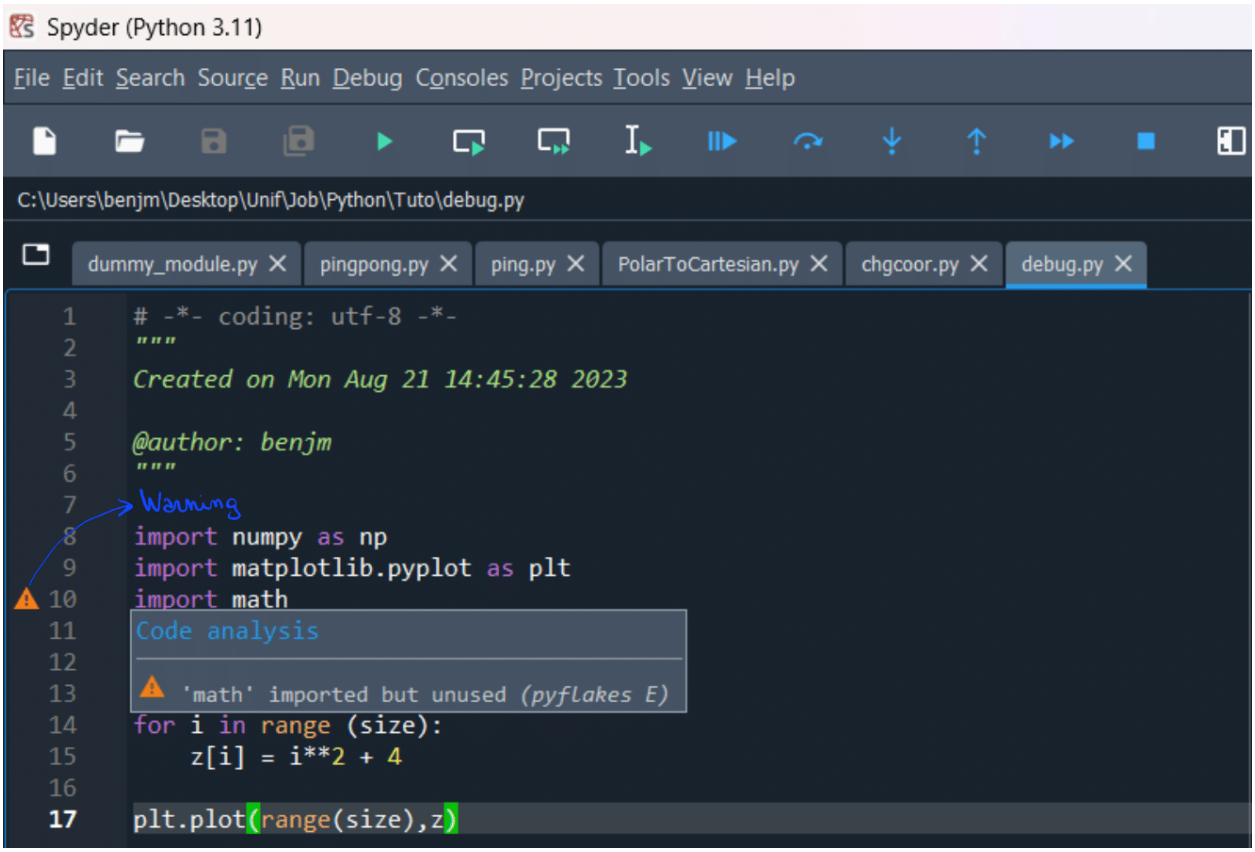
15 Debugging

Spyder includes a debugger integrated into the editor:

https://docs.spyder-ide.org/current/panes/debugging.html

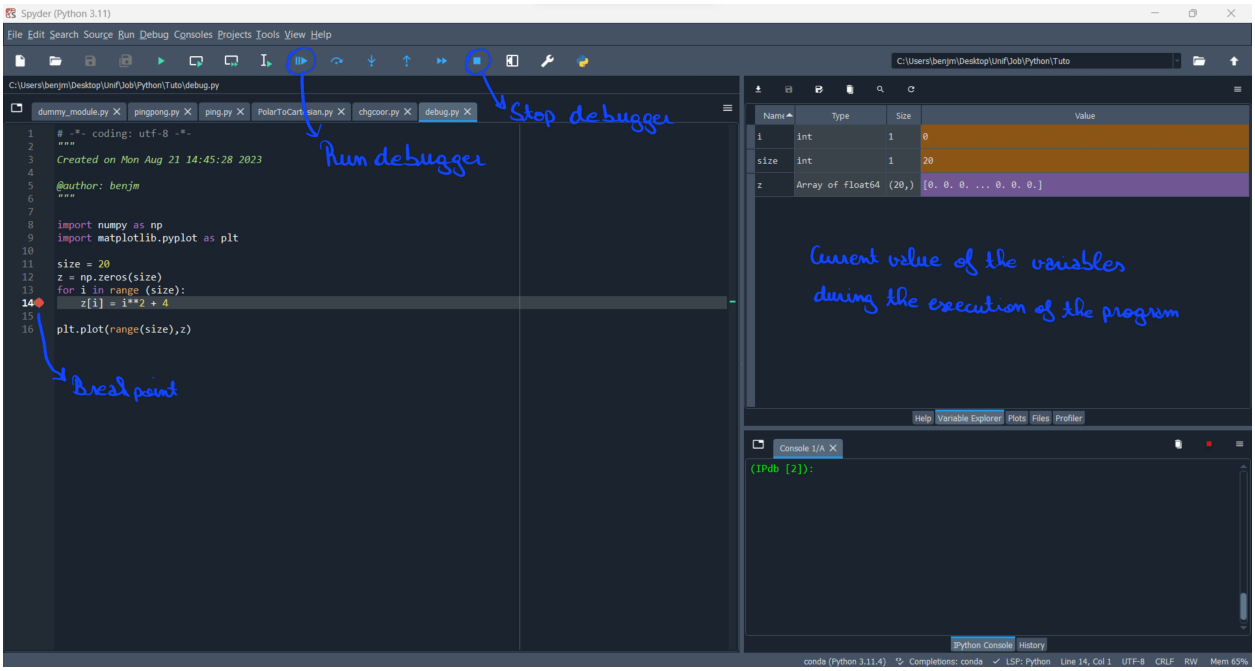
Firstly, when editing a `.py` file, the editor will automatically detect syntax errors, for instance, and will display a red circle with a cross inside to signal the error. This type of critical error will prevent the script from running. The editor can also display a warning icon to indicate, for example, the non-use of an imported library. This is not a critical error, and the program will be able to run.

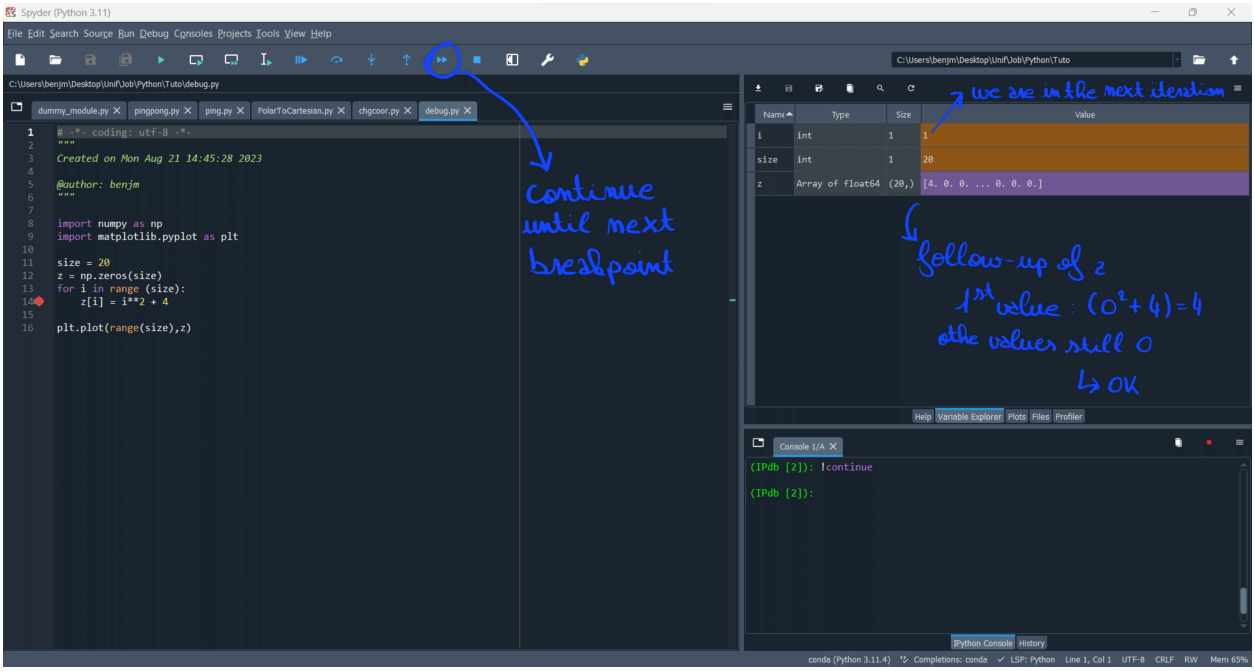




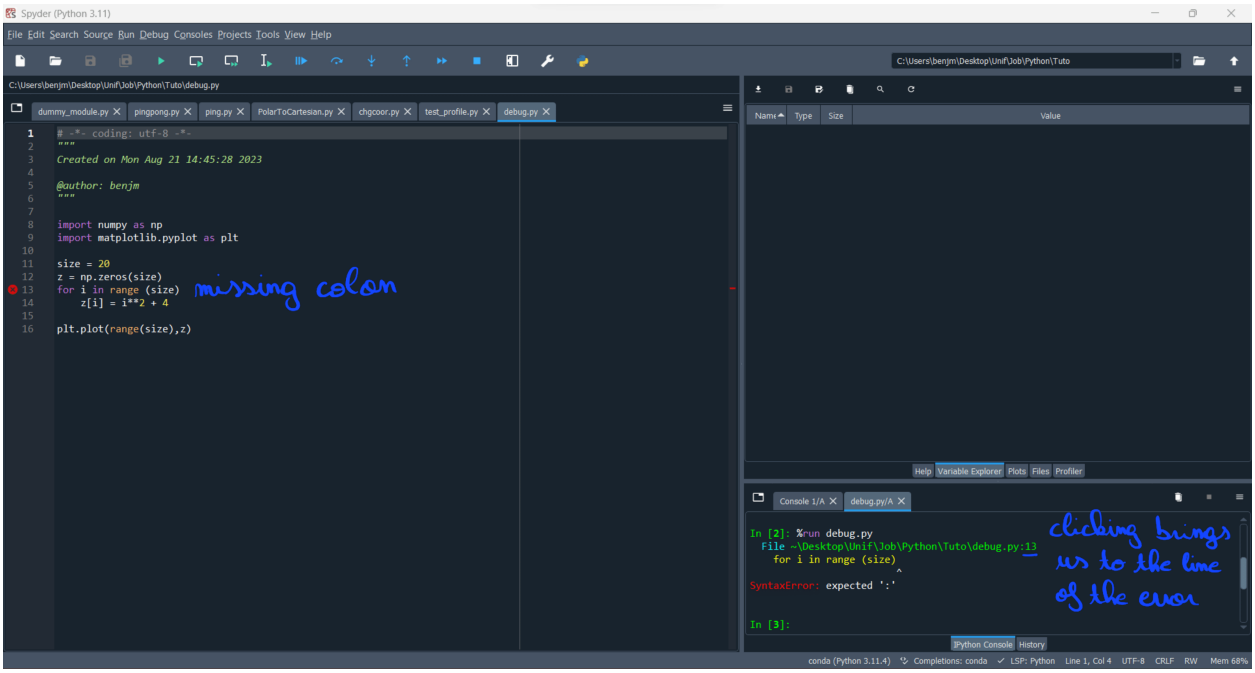
In addition to this real-time verification, Spyder offers a tool that allows stopping a program during its execution to inspect the values of different variables and detect potential errors. For this purpose, breakpoints must be set beforehand in the `.py` file(s). To set a breakpoint, simply click to the right of the corresponding line number, and a red dot will appear to the left of the current line.

The program is then run using the Debug file button. The program stops at the breakpoint indicated by an arrow. You can examine the contents of the variables in the Workspace, step over to the next line, continue execution until the next breakpoint, or stop the program altogether. When a line calls a function, it is possible to step into that function or skip directly to the next line.





Debugging features are also available when a program produces an error. In the console, the error message in red indicates the line where the error occurred. You can directly access it by clicking on the line number.



Exercise: Debug the following code:

In [ ]:
```
%reset
a == 2
b == 3

def sum(a, b)
    c = a + b
    return c

sum(12 14)

if a = 2;
    print(a is equal to two)
else:
print(a, "is different from two")
```

16 Performance Analysis

Advanced performance tools exist for analyzing the efficiency of Python programs: https://docs.python.org/3/library/profile.html

The `cProfile` module allows you to see how much time the executed program spends in each function and the number of function calls:

`ncalls`: the number of calls to a function

`tottime`: total time spent in a function without counting the time spent in calls to sub-functions

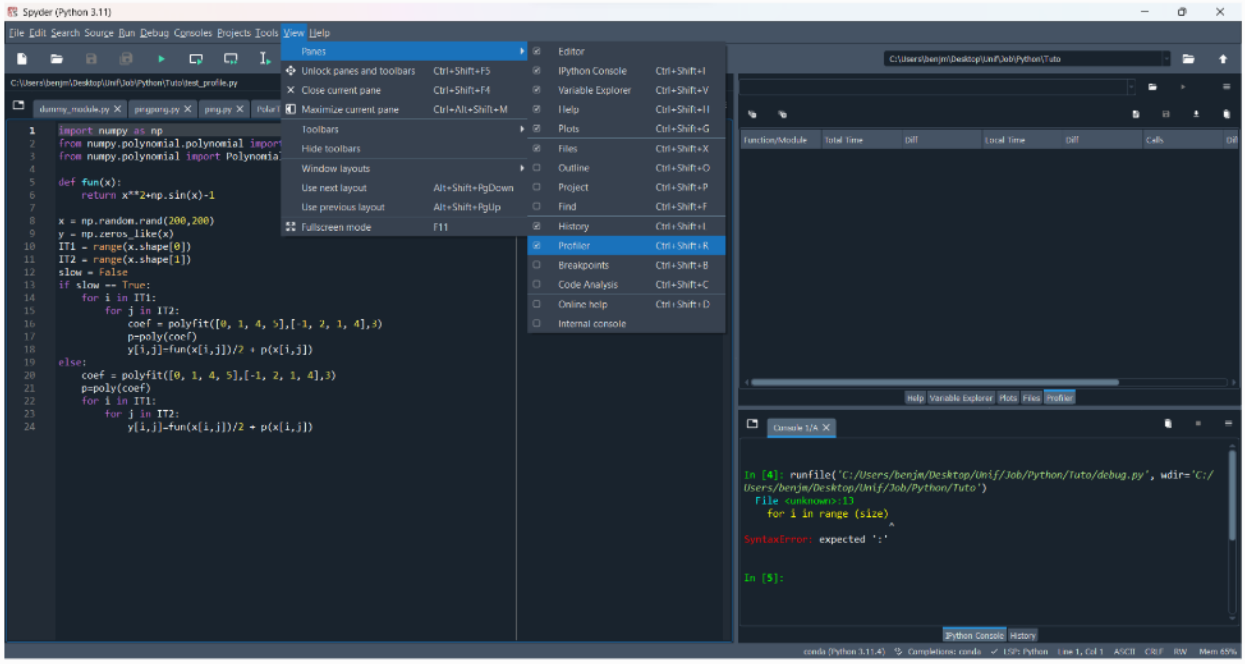
`percall`: time spent per call

`cumtime`: cumulative time spent in the function and sub-functions

`percall`: `cumtime` divided by the number of calls

`filename:lineno(function)`: function data

The Spyder IDE provides a readable graphical interface allowing us to quickly analyze the time taken by our Python scripts. To access it, you first need to ensure you have activated the `Profiler` through the `View > Panes > Profiler` menu.



Now, let's consider the script `test_profile.py` below which includes two loops, a custom function `fun`, and uses functions from the `NumPy` library to interpolate a function defined at several points.

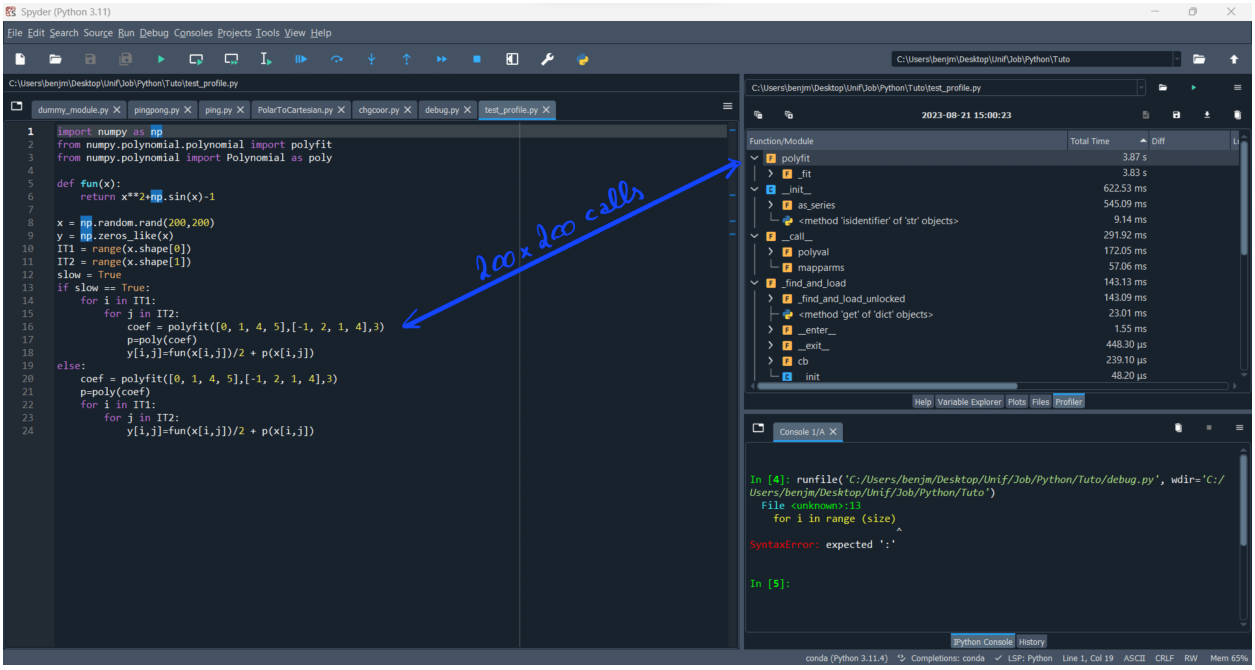
In[154]:
```
import numpy as np
from numpy.polynomial.polynomial import polyfit
from numpy.polynomial import Polynomial as poly

def fun(x):
    return x**2+np.sin(x)-1

x = np.random.rand(500,500)
y = np.zeros_like(x)
IT1 = range(x.shape[0])
IT2 = range(x.shape[1])
slow = True
if slow == True:
    for i in IT1:
        for j in IT2:
            coef = polyfit([0, 1, 4, 5],[-1, 2, 1, 4],3)
```

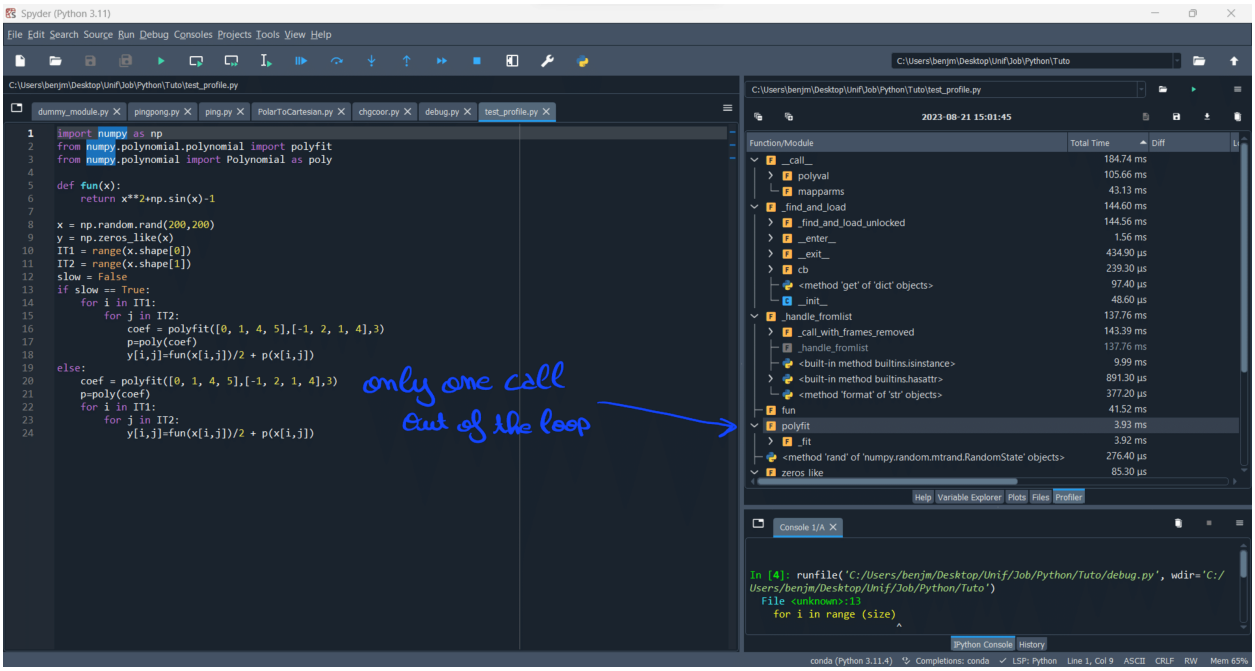
```
            p=poly(coef)
            y[i,j]=fun(x[i,j])/2 + p(x[i,j])
else:
    coef = polyfit([0, 1, 4, 5],[-1, 2, 1, 4],3)
    p=poly(coef)
    for i in IT1:
        for j in IT2:
            y[i,j]=fun(x[i,j])/2 + p(x[i,j])
```

We will test it with the help of the `Profiler` and sort the results by cumulative time to see which part of the code takes the longest to run. Go in the menu `Run > Run profiler`. First, we run a "slow" version of the code with the variable `slow = True` and we get the following result:



We can see that when executing the script with `slow = True`, we calculate the coefficients of a polynomial using `polyfit` each time in both loops, even though these coefficients are independent of the loops. The time taken to call the `polyfit` function 40000 times is 3.87 seconds.

To improve the efficiency of this function, we can move the coefficient calculation out of the loops and compute it only once (`slow = False`). We can then verify that `polyfit` is called only once and that the time taken to execute the script is much shorter.



There are also functions in Spyder to time the execution of Python code, such as `%time` and `%timeit`.

Exercise: Create a script of your choice to compare the use of the `sum()` function from `NumPy` with your own function, which should use a `for` loop to calculate the sum of the elements of a one-dimensional `ndarray`. Analyze the performance of your function compared to the `NumPy` function.

In [ ]:

17 End of the tutorial

We have come to the end of this tutorial. Before diving into Python, try out the various features presented here. Do not hesitate to seek more information whenever you use python.

In[155]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

Print dependencies

In[156]:
```
%load_ext watermark
```

In[157]:
```
%watermark -v -m -p numpy,scipy,matplotlib,time,watermark
print(" ")
%watermark -u -n -t -z
```

```
Python implementation: CPython
Python version       : 3.11.4
IPython version      : 8.12.2
```

```
numpy     : 1.25.2
scipy     : 1.11.1
matplotlib: 3.7.1
time      : unknown
watermark : 2.4.3

Compiler    : MSC v.1916 64 bit (AMD64)
OS          : Windows
Release     : 10
Machine     : AMD64
Processor   : Intel64 Family 6 Model 165 Stepping 2, GenuineIntel
CPU cores   : 12
Architecture: 64bit


Last updated: Mon Aug 21 2023 11:36:05Paris, Madrid (heure d'été)
```