Docs » 1. Introduction to python

1. Introduction to python

```
print("Hello, world!")
```

Python is one of the most used computer languages nowadays. It's an interpreted language, but highly efficient since its libraries typically call Fortran or C codes on their backend. Its Wikipedia article has a bit of its history and links to interesting texts and books.

Here I'll give a brief introduction to essential features of the language and how to use it. I suggest we use the Anaconda distribution, since it runs equally well in Linux, Windows and Mac. Also, besides my introduction below, check also these tutorials:

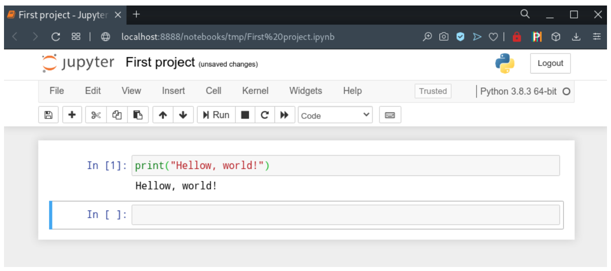
1. Python at Tutorial's point: [python], [numpy], [matplotlib]
2. Official documentation for [numpy - scipy - matplotlib]
3. Official documentation: how to install Anaconda
4. Youtube video (in Portuguese) Como instalar Anaconda em Windows e Linux

1.1. The IDEs

The *Integrated Development Environments (IDEs)* are the softwares that we use to edit / debug / run our codes. Here, I'll suggest we start with these two that are already installed with Anaconda:

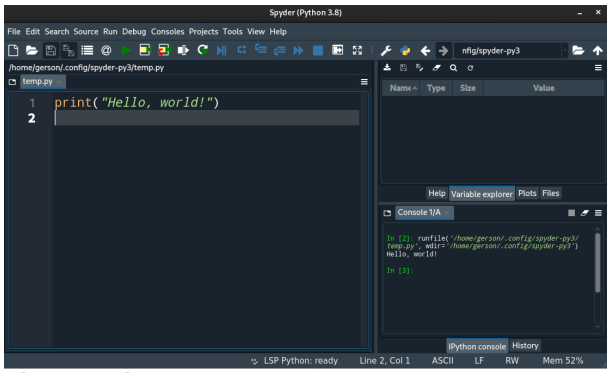
- Jupyter: is a web-based IDE that runs in your browser. It runs the code in cells and allows you to use *markdown* and *Latex* to document your code. Since *jupyter* integrates code, documentation and images, I **recommend** we start with this one. The name **jupyter** refers to the three initial languages it was design to support: *julia*, *python*, and *R*.
- Spyder IDE: is a traditional IDE with the text editor and dedicated panels for the output and figures. The name **Spyder** means *Scientific Python Development Environment*. The figure below shows how it looks like.

The images below show how these IDE look like, click to zoom.



```
In [1]: print("Hellow, world!")
        Hellow, world!
In [ ]:
```

The Jupyter IDE



```
print("Hello, world!")
```

```
In [2]: runfile('/home/gerson/.config/spyder-py3/temp.py', wdir='/home/gerson/.config/spyder-py3')
Hello, world!

In [3]:
```

The Spyder IDE

Other very interesting IDEs are the PyCharm and VS Code. I actually use **VS Code** for everything: python, C, and even for writting this webpage. But if you are a beginner, I recommend you first try **Spyder** or **Jupyter**.

Note

Jupyter vs Spyder: Both are great IDEs, however, they are quite different from each other. Their main *pros* and *cons* are:

- *Jupyter*
  - (pro) Saves in a single *notebook* the code, markdown/latex notes, and the figures/plots;
  - (pro) Run code in cell blocks that allow you to split calculations and plots;
  - (con) The *notebook* (.ipynb) is not a simple text file, but a *json* file;
  - (con) There's no structure for debugging.
- *Spyder*
  - (pro) Saves the code as simple text (.py);
  - (pro) It has a native debugging structure;
  - (pro) Allows you to run individual *cells* (similar to jupyter);
  - (con) Figures must be saved into files for storage.

1.1.1. Using the IDEs

More details will be covered in the first class with simple examples, but here are the most important things to remember.

- **Jupyter:** The code runs in blocks called *cells*. To **run a cell** type `SHIFT+ENTER`. The jupyter codes are saved in a *json* format (extension .ipynb) and are called **ipython notebooks**, which allows you to save the code, markdown and images in a single file.

Warning

Evidently, you can run the cells out of order while developing, but the final code should be organized to run cell-by-cell in order.

The following **keyboard shortcuts** run in *Command mode*, which is activated by pressing `ESC`. A full list of shortcuts can be seen by pressing `ESC+H`. To **add a cell**, type `ESC+A` to add above or `ESC+B` to add below. To **delete a cell** type `ESC+D+D`.

To **change a cell type** check the buttons bar at the top of the window you see a list of cell types. The most important ones are *code* and markdown. In the *markdown mode* you can write formatted text and Latex equations, which is quite useful to document your code and present the results. You can also change the cell to *markdown mode* by calling `ESC+M`, and back to *code mode* with `ESC+Y`.

- **Spyder:** This IDE works with simple text files (extension .py). To run the (entire) code you press `F5`. Spyder can also work with cells, which are defined by `# %% Cell Name`. The *Cell Name* is optional, but helps with the organization of the code. To run only the current cell and advance to the next, you press `Shift + Enter`.

Warning

It might be important to check the **workspace directory** on the top right corner of the window, as it **defines the relative path** to read and save files.

Spyder allows you to choose how to show the plots/images. To find the options go to `Tools > Preferences > IPython console > Graphics > Backend`. To capture the plots into the Spyder panel, choose **inline**, and to plot as a separate window, choose **automatic**.

1.2. Variables, data types and operations

For those used to C and Fortran, it seems weird that **variables don't need to be declared**. Their type is inferred by the assignment, which can be checked using the `type(...)` call. Try running these examples

Example: Assignment and data types

```python
# These will all be integers
a = 3
b = 4
c = a**2 + b**2
print('type of a is', type(a))
print('type of b is', type(b))
print('type of c is', type(c), ' and its value is', c)

# These are floats (floating point, real numbers)
x = 1/3
y = 4.2
z = a/b
Na = 6.022e23 # here 1e23 = 10²³
print('x =', x, ' has type', type(x))
print('y =', y, ' has type', type(y))
print('z =', z, ' has type', type(z))
print('Na =', Na, ' has type', type(Na))

# For complex numbers, use j instead if i
c = 4.5 + 3.1j
d = 2 + 3j
print('c =', c, 'has type', type(c))
print('d =', d, 'has type', type(d))

# Strings
s = 'hello world!'
print(s, 'is a ', type(s))

# Mixing strings and numbers with the str(...) cast
#   1st, using an integer to label a file
n = 8
myfile = 'somefile' + str(n) + '.txt'
print('File name:', myfile)
#   2nd, now using a float, but rounding it up
x = 1/3
myfile = 'somefile' + str(round(x, 2)) + '.txt'
print('File name:', myfile)
```

In the last lines above we use the `str(...)` call to cast its argument into a string. You can also use casting to integers with `int(...)` or floats with `float(...)`. Try it! Above we use `round(..., n)` to trim a number up to *n* digits.

1.2.1. Arithmetic operations

Check these examples:

Example: arithmetic operations

```python
# let's start assigning numbers to x and y
x = 5
y = 2

# and now let's operate and print the results
print('addition: ', x + y)
print('subtraction: ', x - y)
print('multiplication: ', x * y)
print('division: ', x / y)
print('exponentiation: ', x**y)
print('remainder: ', x % y)
```

Besides the simple assignments, python allows for increments. For instance, `x += 2` is the same as `x = x + 2`. Try these examples:

Example: assignment with increments

```python
# let's start with a simple assignment
x = 2
# and apply the increments

x += 5 # the same as x = x + 5
print('now x =', x)

x -= 5 # the same as x = x - 5
print('now x =', x)

x /= 2 # the same as x = x / 2
print('now x =', x)

x *= 2 # the same as x = x * 2
print('now x =', x)
```

Notice in the example above that the type of x has changed at some point. **Why?**

1.2.2. Comparisons and logical operations

Comparisons operations are simple `==`, `>`, `<` and etc. The logical operations act on `True` or `False` values by combining it with `and`, `or`, `not` operations. We'll see how to use comparisons with `if` and loops later. For now, let's check the examples:

Example: Comparisons and logical operations

```python
x = 3
y = 4
z = 3

# simple comparisons
print('is x larger than y?', x > y)
print('is y larger than z?', y > z)
print('is x larger or equal to z?', x >= z)
print('is x different than z?', x != z)
print('is z equal to z?', x == z)

# composed comparisons
print('is x larger than both y and z?', x > y and x > z)
print('is x larger or equal to z?', x > z or x == z) # the same as x >= z
print('is x between 1 and 7?', 1 <= x <= 7)
```

Comparisons and floats: BE VERY CAREFUL! You should **NEVER** use `==` to check equivalence between floats, and this example shows why:

Example: Error comparing floats

```python
x = 0.1
y = 3 * x
z = 0.3

# should both questions be True?
print('obviously z == 0.3 by definition, right?', z == 0.3)
print('and y is also 0.3, right?', y == z)

# let's check the values
# the format call allows you to specify the number of digits
print('x = ', format(x, '0.30f'))
print('y = ', format(y, '0.30f'))
print('z = ', format(z, '0.30f'))
```

What's happening there? Shouldn't both be 0.3???? While 0.1 and 0.3 are exact in base 10, in binary they are repeating fractions: $(0.3)_{10}$ = $(0.0[1001])_2$, and $(0.1)_{10}$ = $(0.0[0011])_2$. The numbers between [...] are the repeating pattern. Since numbers are stored in memory with 64 bits (typically), it requires a truncation. For instance, if you truncate after three repetitions, the 0.1 becomes $(0.0001100110011)_2$ = $(0.0999755859375)_{10}$, and the 0.3 is $(0.0100110011001)_2$ = $(0.2999267578125)_{10}$.

1.2.3. Lists and dictionaries

A list in python is indeed a list of **whatever elements**. You can mix oranges and bananas… and numbers as well. This is different from an *array*, which is a structure with a well defined type and we'll discuss within the *numpy* section. Let's focus on generic lists for now, check the example:

Example: lists and operations on lists

```python
# using only strings for now
cart = ['banana', 'oranges'] # init list with two items
cart.append('apple') # add an item
cart.sort() # sort alphabetically

print('Number of elements:', len(cart)) # len from length
print('The first item: ', cart[0]) # indexes start from 0
print('The last item: ', cart[-1]) # and you can count backwards
print('Is there bananas?', 'banana' in cart) # a membership comparison
```

Above we have used `in`, which is a **membership comparison** and be used with `if` and loops below.

Now let's start with an empty list and mix types as we add to the list:

Example: mixing types

```python
mylist = [] # start empty
mylist.append(2) # add an integer
mylist.append(2.0) # and an float
mylist.append('two') # add an string

print('The list:', mylist)
```

Warning

A list is not a mathematical vector, it does not support mathematical operations. For instance, the code `x = [1, 2, 3]` and `y = 2*x` **WILL NOT GIVE** `[2, 4, 6]`, instead, it repeats the list twice to give `[1, 2, 3, 1, 2, 3]`. This is useful to create long lists with a repeated pattern.

The mathematical vectors will be defined within the **numpy** package as `x = np.array([1, 2, 3])`. In this case the call `y = 2*x` will return `[2, 4, 6]`.

1.2.3.1. Dictionaries

It can be useful to use a list of mixed types to store different parameters for your code. But it's even better to use **dictionaries**. It basically works like a list, but instead of using integers 0, 1, … as indexes, it uses strings or integers. Let's check:

Example: using dictionaries to store parameters

```python
pars = {} # init an empty dictionary

pars['T'] = 273 # K
pars['P'] = 1.013e5 # Pa
pars['V'] = 22.4 # L
pars['filename'] = 'myfile.txt'
pars[5] = 'five' # useless example as an example

# let's change a value
pars['T'] = 300

# and print all
print('Temperature is', pars['T'])
print('Pressure is', pars['P'])
print('Volume is', pars['V'])
print('Store in file: ', pars['filename'])
print('Element 5 is: ', pars[5])
```

Notice above that we can use an integer 5 as the index, but it is not as useful as using strings. The idea is to use it when your code has many parameters, and it's easier to pass it around as a dictionary instead of using many… many arguments on each function. Use it wisely!

1.3. Decision making: if / else / etc…

Using `if/else` is as simple as in any other language. We just need to check the syntax. But remember that we can use the simple comparisons `>, <, >=, <=, !=`, the logical operators `and, or, not` and membership operators `in, not in`. Let's check it:

Example: if / else and comparisons

```python
# let's start with a simple one
a = 3
b = 4
if a > b:
    print('a is larger than b')
elif a < b:
    print('b is larger than a')
else:
    print('they are equal')

# now let's check a membership comparison with lists
cart = ['apple', 'banana', 'orange']

if 'grape' in cart:
    print('yes, we have grapes')
else:
    print('no, we need grapes')
```

Notice that the structure has no termination. The segment is delimited by the **indentation**.

1.4. Loops: for / while and comprehensions

The loop **for** is usually used when it runs over a predefined *list* of elements, while the **while** uses a less predictable termination point. Let's start with the **for**:

Example: using a for loop over lists

```python
# let's use our fruits again
cart = ['apple', 'banana', 'orange']
for fruit in cart:
    print('we have:', fruit)

# similarly, you could also do
for i in range(len(cart)):
    print('item', i, ' is', cart[i])
```

So you can loop over the elements of a list using the membership operator `in`, or you can use an integer `i` to loop over the indexes. In this case we use `len(cart)` to get the number of elements in the list (3) and the command `range(...)` to create a list of integers. Let's check how **range** works in this example:

Example: using range

```python
# range(n) = 0..n-1
for i in range(10):
    print(i)

# range(ni, nf) = n1..nf-1
for i in range(3, 15):
    print(i)

# range(n1, nf, step) takes steps instead of increasing by 1
for i in range(1, 15, 2):
    print(i)
```

Warning

Notice that **range** defines an interval closed at the left side and open at the right side.

`range(init, end, step)` goes from **init** to **end-1** in steps of **step**

You can use any type of *lists* or *arrays* (*numpy*) to delimit the for loop.

Now let's check a **while** example:

Example: using while to sum 1 + 1/2 + 1/4 + 1/8... until the new element is small enough

```python
x = 1 # init x
s = 0 # and init the sum
# loop until x is small enough
while x > 1e-5:
    s += x # add to the sum
    x /= 2 # update x

# print the results
print('the final x =', x)
print('the sum s =', s)
```

Above we are not specifying the number of loops, but looping until x gets small enough.

A **compact for loop** can be used to define lists as **comprehensions**. Check this example:

Example: comprehensions

```python
# let's start with a list for the example
x = [0, 2, 4, 5, 9]

# and define y using a comprehension:
y = [xi**2 for xi in x]

print('x = ', x)
print('y = ', y)
```

The **comprehension** executes the code before the *for* for each element in the list.

1.5. Functions and scope of variables

As usual, functions take arguments and returns something. The main differences from C/Fortran is that a function can return more than one object. Also, there's a **compact form** for inline functions called **lambda** functions. Let's check the examples:

Example: simple functions

```python
# import the square root from the complex math library
from cmath import sqrt

# define the function
# here c has a default value
def bhaskara(a, b, c=0):
    d = sqrt(b**2 - 4*a*c)
    x1 = (-b+d)/(2*a)
    x2 = (-b-d)/(2*a)
    return x1, x2

# calling the function
s1, s2 = bhaskara(1, 2, -15)
print('sols:', s1, 'and', s2)

# let's call again with different numbers
x1, x2 = bhaskara(1, 5, 0)
print('sols:', x1, 'and', x2)

# above, we could have omitted c
x1, x2 = bhaskara(1, 5)
print('sols:', x1, 'and', x2)

# a final example
x1, x2 = bhaskara(1, 2, 2)
print('sols:', x1, 'and', x2)
```

Above we are **importing** the `sqrt` from the *cmath* library to allow for complex numbers.

The function *bhaskara* receives three parameters, but the third one has a **default keyword argument**. If not informed, it's assumed to be zero as indicated. At the end, the function returns two values, x1 and x2, which are attributed to two variables on the calls that follow.

1.5.1. The scope

Notice above that we have variables x1 and x2 within the function bhaskara, and also outside the function. These are not the same variables. The (x1,x2) variables within bhaskara have the same name but are not the same variables as those (x1,x2) outside. To make it clear, let's try this other example, which you could run in **debug mode** to follow the values of the variables.

Example: scope of a variable

```
 1  # some random function
 2  def f(x):
 3      a = 10
 4      s = a * x**2
 5      return s
 6
 7  # let's call it directly
 8  print('the value of f(10) is', f(10))
 9
10  # is x, a, or s defined?
11  print('x =', x)
12  print('a =', a)
13  print('s =', s)
14
15  # let's try again
16  a = 0
17  print('is f(10) now zero?', f(10))
18
19  # let's define x
20  x = 5
21  print('x is now = ', x)
22  print('for this x we have f(x)=', f(x))
23
24  # let's call f(10) again
25  print('the value of f(10) is', f(10))
26  print('did it change the value of x? x=', x)
```

We have to understand the **local scope** of each variable:

Within the function `f(x)`, its argument *x* and the inner variables *a* and *s* are **local variables**, their value and definition are set only within the function and are not accessible outside unless you **return** their values. On the outside, the *a* and *x* defined in lines 16 and 20 are on the **global scope**.

1.5.2. The lambda functions

A *lambda* function is simple a short notation for short functions:

Example: the lambda functions

```python
# let's start by defining a very simple function
def f(x, y):
    return x**y

# which can be defined also as a lambda function
g = lambda x, y: x**y

# let's compare:
print('calling f:', f(2, 10))
print('calling g:', g(2, 10))
```

The *lambda function* is defined such that the arguments follow the *keyword* **lambda** and the direct return follows after the **:**

1.6. Importing libraries

The core of python comes with many functionalities, but it always need to be complemented with external libraries using **import** as shown above for **cmath**. There are many ways to import a library.

You should avoid importing like this:

Example: causing a conflict with BAD IMPORTS

```python
# this imports only the sqrt command from math
from cmath import sqrt

# this imports sqrt from math (non-complex math library)
from math import sqrt

# you could, but shouldn't import everything as well
from math import *
```

Warning

Notice that by calling `from math import *` or the other examples above you may cause conflicts since the sqrt function exists in more than one library. The correct way is shown next.

The proper way to import a library is:

Example: safe import

```
import math as rm
import cmath as cm
# the import alias do not have to be rm and cm, you can choose whatever (?) you want

# using the real math library
print('The square root of 2 is: ', rm.sqrt(2))

# using the complex math library
print('The square root of +2 is: ', cm.sqrt(+2))
print('The square root of -2 is: ', cm.sqrt(-2))
```

It's a pain to carry the *objects* `rm.` or `cm.` up and down the code, but that's the safe way and you should use it!

Above we use *rm* and *cm* as alias to make the calls shorter, but you could have also used simply `import math` without an alias. In this case the calls would be `math.sqrt(2)` and so on.

In practice, always try to use common alias for the libraries, for instance, we'll use these a lot:

Example: common libraries and their alias

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
```

1.7. Reading/saving files and string manipulation

To save data files it might be better use the **numpy** package. The discussion here will be more useful to read or save structured files. An important example would be to read a file with parameters for your code.

Commands to discuss:

- open and close, using with
- read / readline
- write / writelines
- `print(data,  file=open(filename, 'w'))`
- find / replace

- split into list
- ...