SIGNAL PROCESSING AND COMPUTING

PYTHON 101

APRIL 29th, 2016

Youri Poulin, Renato Borges, Florian Fauchier,
Yannick Roehlly, Alain Schmitt, Christian Surace

PYTHON

IMPORT

- Populates main namespace (Please, don't):

```
from os import *
path.join("/folder/", "file")
```

```
from os.path import *
join("/folder/", "file")
```

```
from os.path import join
join("/folder/", "file")
```

```
from os import path
path.join("/folder/", "file")
```

- Dedicated namespace:

```
import os
os.path.join("/folder/", "file")
```

```
import os.path
os.path.join("/folder/", "file")
```

IMPORT

- Dedicated customized namespace (recommended):

```
import os.path as osp
osp.join("/folder/", "file")
```

```
from os import path as p
p.join("/folder/", "file")
```

```
import numpy as np
np.array([1, 2, 3])
```

TYPES

MAIN TYPES

- Numerical:
  - int : 42
  - float : 42.
- Iterables::
  - str : "The answer is 42" tuple : `("The answer", "is", 42)`
  - list : `["The answer", "is", 42, ["a list"]]`
  - dic : `{"The Question" : "the ultimate one", "The answer:" : 42}`
- Logical:
  - bool: `True`

LISTS

- Create a list:

```
>>> x = [4, 12, 3.2, 45, 11]
>>> print(x)
[4, 12, 3.2, 45, 11]
```

- Index positions:

```
Index from rear:    -6  -5  -4  -3  -2  -1
Index from front:    0   1   2   3   4   5
                   +---+---+---+---+---+---+
                   | a | b | c | d | e | f |
                   +---+---+---+---+---+---+
Slice from front:  :   1   2   3   4   5   :
Slice from rear:   :  -5  -4  -3  -2  -1   :
```

LISTS

```
>>> print(x)
[4, 12, 3.2, 45, 11]
```

- Access with index:

```
>>> x[0]
4
```

```
>>> x[1:]
[12, 3.2, 45, 11]
>>> x[1:3]
[12, 3.2]
>>> x[-2:]
[45, 11]
```

TUPLES

- Create a tuple:

```
>>> x = (4, 12, 3.2, 45, 11)
>>> x = 4, 12, 3.2, 45, 11    # equivalent
>>> print(x)
(4, 12, 3.2, 45, 11)
```

- Assign a tuple to multiples variables:

```
>>> a, b, c = (4, 12, "something")
>>> print("a = {}; b = {}; c = {}".format(a, b, c))
a = 4; b = 12; c = something
```

NUMPY

NUMPY ARRAYS

- Array from list (or tuple):

```
>>> ar1 = np.array([1.23, 4.67, 5.8, 7.2])
>>> print(ar1)
[ 1.23  4.67  5.8  7.2 ]
```

- Operation on arrays:

```
>>> ar2 = np.array([1, 2, 0, 10])
>>> ar1*ar2
array([  1.23,   9.34,   0.  ,  72.  ])
>>> ar1*10
array([ 12.3,  46.7,  58. ,  72. ])
```

NUMPY

- Arrays comparison :

```
>>> ar1 < ar2
array([False, False, False,  True], dtype=bool)
```

- Where function :

```
>>> np.where(ar1>ar2)
(array([0, 1, 2]),)
```

- Cast an array:

```
>>> ar2
array([ 1,  2,  0, 10])
>>> ar2.astype(float)
array([  1.,   2.,   0.,  10.])
```

NUMPY

- ⚠ Slicing a numpy arrays returns a view.
  ⇒ Modifying this view will modify the original array! :

```
>>> print(ar1)
[  1.23  4.67     5.8    7.2 ]
>>> v = ar1[1:]
>>> v[0] = 999
>>> print(ar1)
[  1.23  999.   5.8    7.2 ]
```

TYPING

- Dynamic typing:

```
>>> x = 145.
>>> type(x)
<class 'float'>
>>> i = 2
>>> type(i)
<class 'int'>

>>> s = "23.2"
>>> type(s)
<class 'str'>
>>> s = float(s)
>>> print(s)
23.2
>>> type(s)
<class 'float'>
```

SYNTAX

- No `{`, no `;`
- Instructions are separated by end of lines
- Structures are delimited by indentation

STRUCTURES

FOR LOOP

- Iteration on an iterable:

```
myList = [2, 5, 3, 7]
for element in myList:
    print(element)
```

```
myList = [2, 5, 3, 7]
for i in range( len(myList) ):
    print( myList[i] )
```

Output:

```
2
5
3
7
```

TEST

- Booleans

```
>>> myBoolean = 1==1
>>> print(myBoolean)
True
>>> not myBoolean
False
>>> myBoolean = 1!=1
>>> print(myBoolean)
False
```

- If:

```
if myBoolean:
  print("Something")
```

```
if ( a == b) and ( ( a != c ) or ( e < f ):
  print("Something")
```

WHILE

```
k = 0
while k <= 3:
    print(k)
    k+=1
```

Output:

```
0
1
2
3
```

FUNCTIONS

- Define a function:

```
>>> def f(x, a = 0.2):
...     res = x**2 + 4.5*x + a
...     return res
>>> y = f(3)
>>> print(y)
22.7
>>> y2 = f(3, a = 0.4)
>>> print(y2)
22.9
```

FUNCTIONS

- Routines are functions:

```
def myRoutine():
    t = 12
    global u
    u = u*100
t = 1
u = 2
myRoutine()
```

```
>>> print(t)
1
>>> print(u)
200
```

FORMAT STRINGS

```
>>> s = "The {} is {}.".format("answer", 42)
>>> print(s)
The answer is 42.
```

```
>>> print( "a = {1}, b = {2}, c = {0}".format(2, 0, 1) )
a = 0, b = 1, c = 2
```

```
>>> t = [1.1, 2.2, 3.3]
>>> "First : {0[0]}, second : {0[1]}, third : {0[2]}".format(t)
'First : 1.1, second : 2.2, third : 3.3'
```

READ A FILE

Basic way:

```
fname = open('example.txt', 'r')
< operations on fname >
fname.close()
```

More secured way:

```
with open("example.txt", "r") as fname:
    out  = fname.read()          # string containing the entire fname
    line = fname.readline()      # One line, goes to the next line
                                 # at each use
    out  = fname.readlines()     # list (one line for one line in the file
```

READ A FILE

Numerical data files

```
>>> ar = np.loadtxt('example.dat', dtype=<type 'float'>,
                    comments='#', delimiter=None)
>>> print(ar)
array([[  1.034   2.2  23.34  ]
       [ 33.2    44.1  43.88  ]
       [ 13.5   673.   14.42  ]
       [ 33.0    67.92  1.88  ]
       [ 83.2    45.    8.    ]
       [ 55.9     3.    9.72  ]]
```

WRITE A FILE

- Classical secured way:

```
with open("exampel.txt", 'w') as fname:
    fname.write("This is the file content\nbye")
```

- Numerical data files:

```
arr = np.array([[1.1, 2.2, 3.3], [2.1, 2.2, 2.3]])
np.savetxt("output.dat", arr, fmt='%.18e', delimiter=' ', newline='\n', h
```

WRITE YOUR PROGRAM/MODULE

```
#!/usr/bin/env python3

import numpy as np

''' _________________   Functions   _________________'''

def datafile_function(input_fname, output_fname, verbose = False):
    input_arr = np.loadtxt(input_fname)
    if verbose:
        print("Input array has been loaded.")
    output_arr = operation_on_array(input_arr)
    np.savetxt(output_fname, output_arr)
    if verbose:
        print("Output array has been written.")

def operation_on_array(input_array):
    return  input_array*10 + 2
```

SUBPROCESS AND MODULES

- Get stdin and stoud from system command:

```
import subprocess as sub
(output, error)=sub.Popen("ls", stdout=sub.PIPE,
                          shell=True).communicate()
```

- System commands and path manipulation:

```
import os
>>>os.getcwd()
'/home/username
>>>os.path.abspath("folder")
'/home/username/folder'
>>>path = os.path.join("/home/username", "folder2")
>>>path
'/home/username/folder2'
>>>os.chdir(PATH)
>>>os.getcwd()
'/home/username/folder2'
```

DOCUMENTATION

https://docs.python.org/3/ text
https://docs.python.org/3/ text

INSTALLING PYTHON

AIM

Have Python installed on your computer and **understand** how this installation works, how to install new packages, how to make updates, etc.

OUTLINE

- Install Python with Anaconda
- The *conda* utility
- The *pip* utility
- Quick overview of environments
- The interactive Python
- The notebook

INSTALLING PYTHON WITH

ANACONDA®

ANACONDA

- Anaconda is a **Python distribution** by *Continuum analytics*.
- Works on GNU/Linux, OSX™©, and Windows™©.
- Contains a lot of scientific packages (in particular astropy).
- Installs Python in a user owned directory without interfering with the system Python.

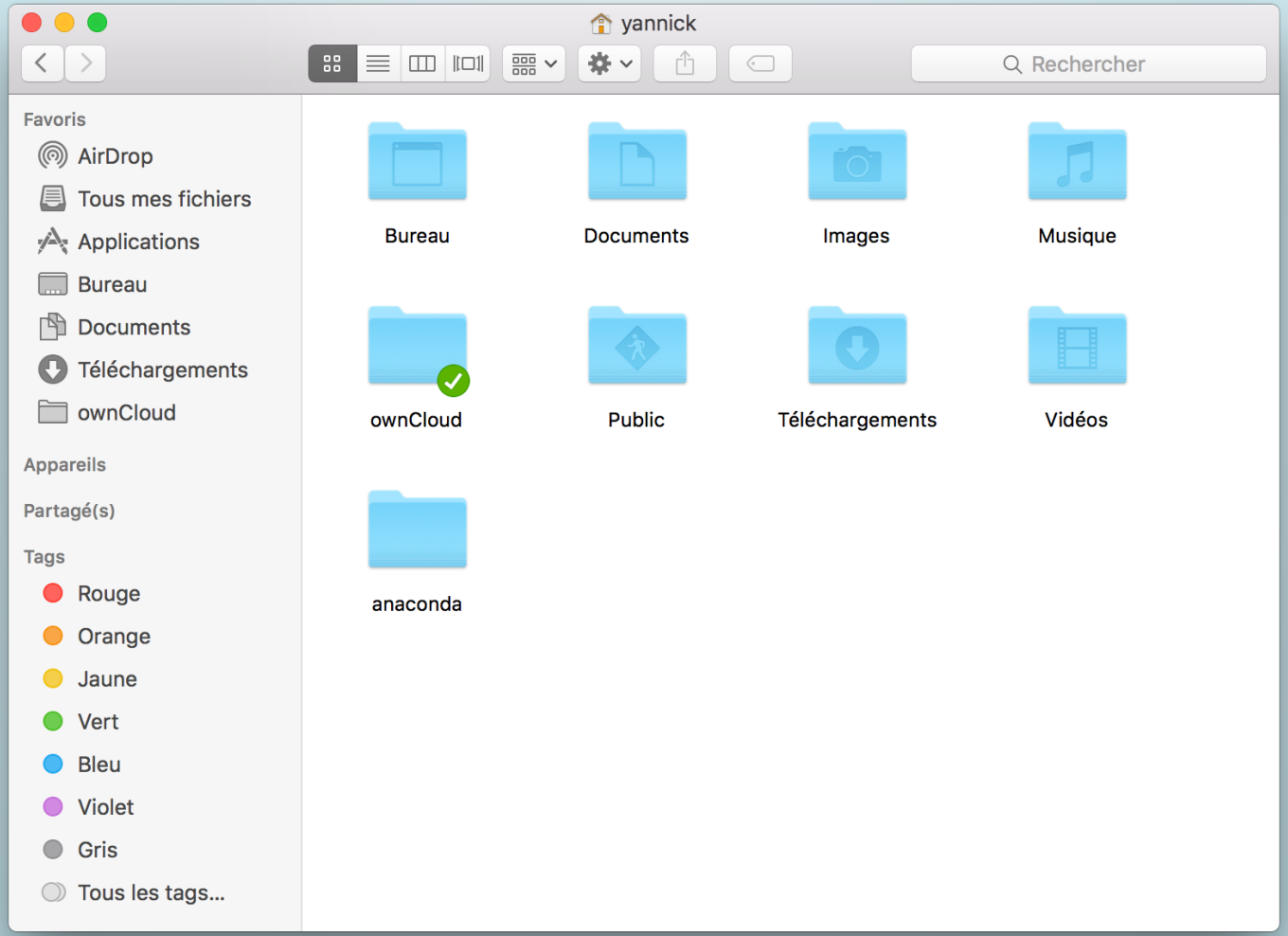
https://www.continuum.io/downloads

INSTALLATION

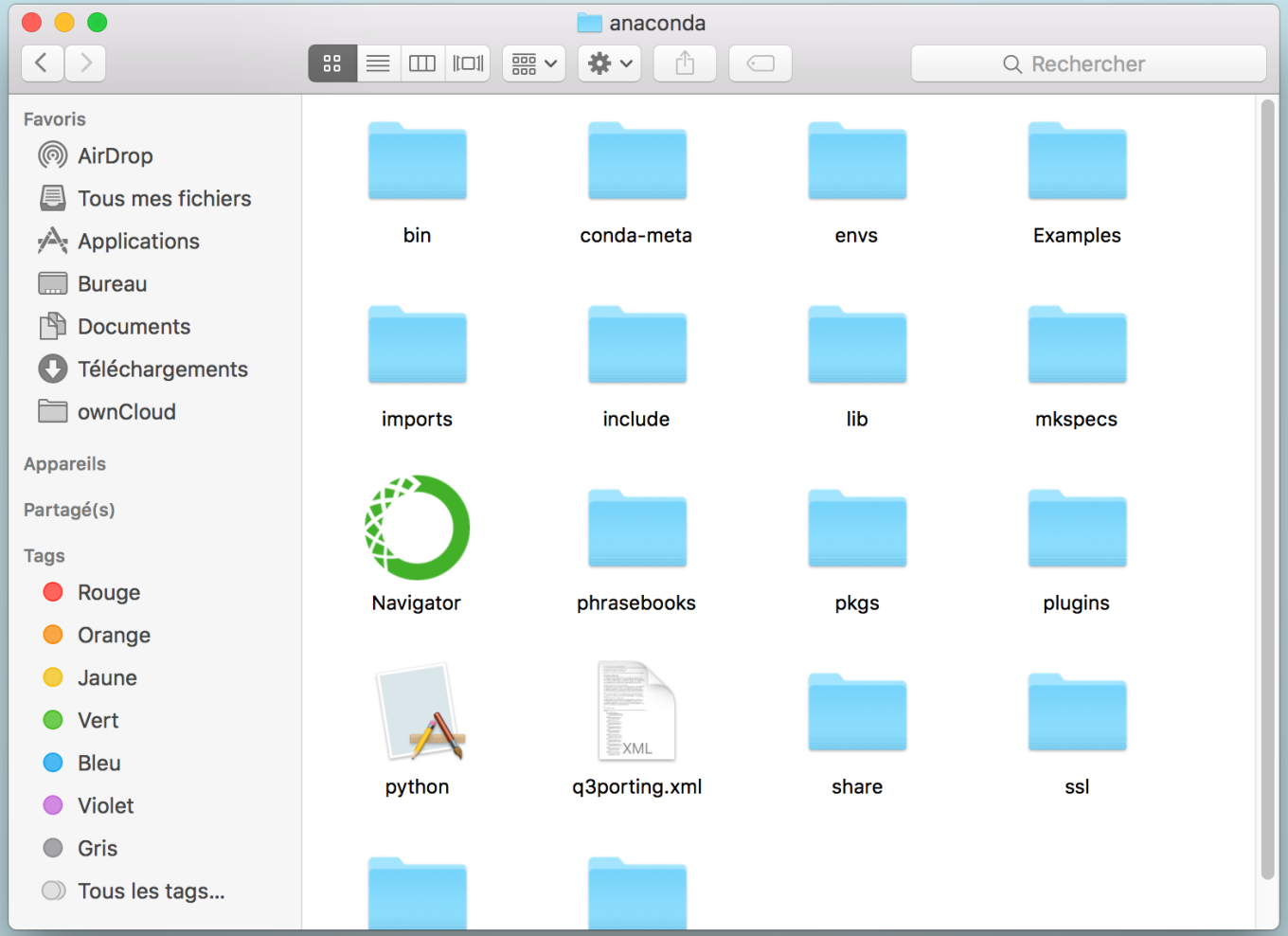
Download the Anaconda installer (take the **Python 3** version) and execute it. In doubt, use default choices.

https://www.continuum.io/downloads

The installer has created an *anaconda* directory in your home folder...



… containing all the distribution, in particular a *bin* folder.

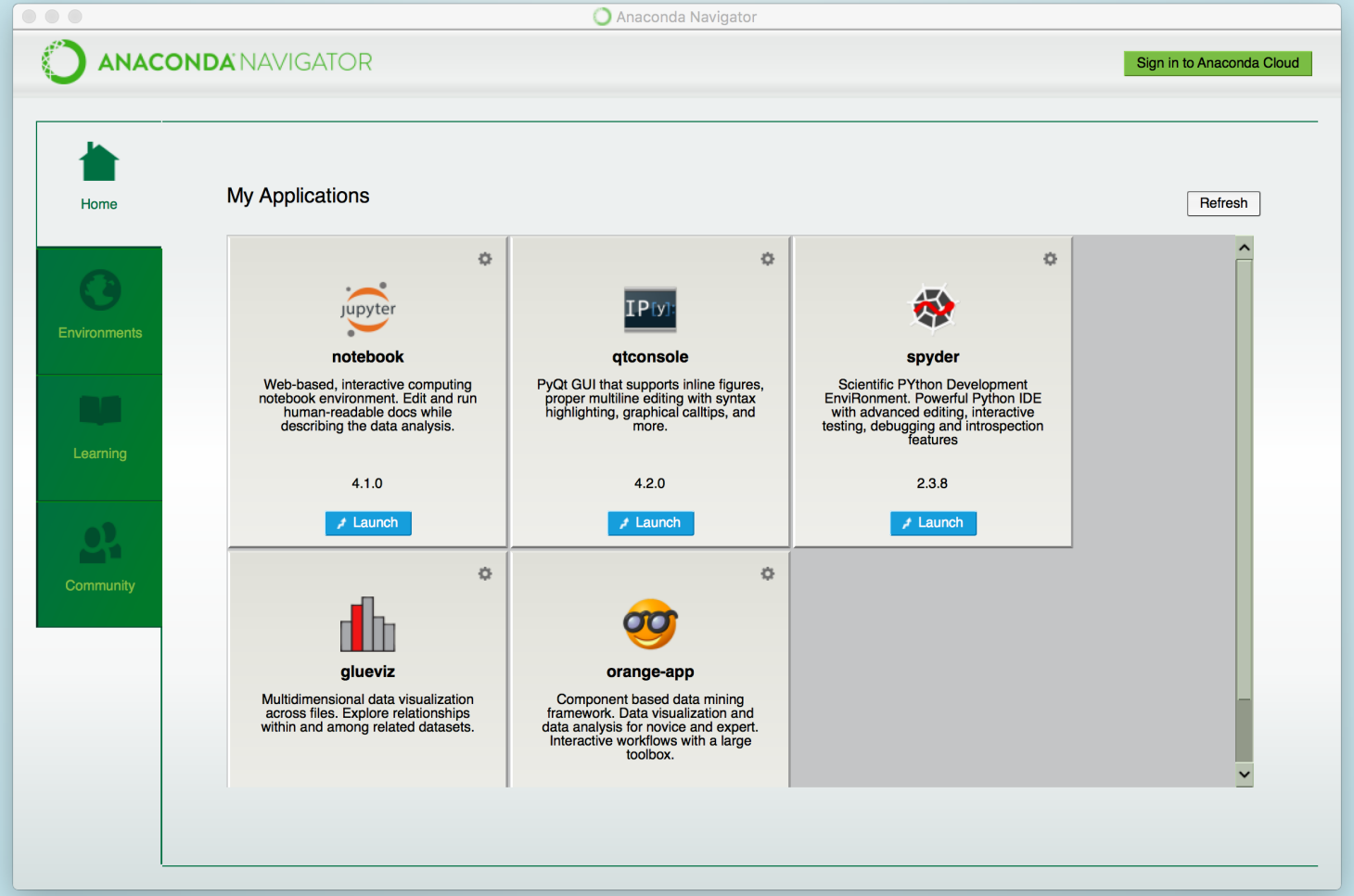


- The installer has also modified the *profile* file to add this *bin* directory in the PATH, **before** your system default path.

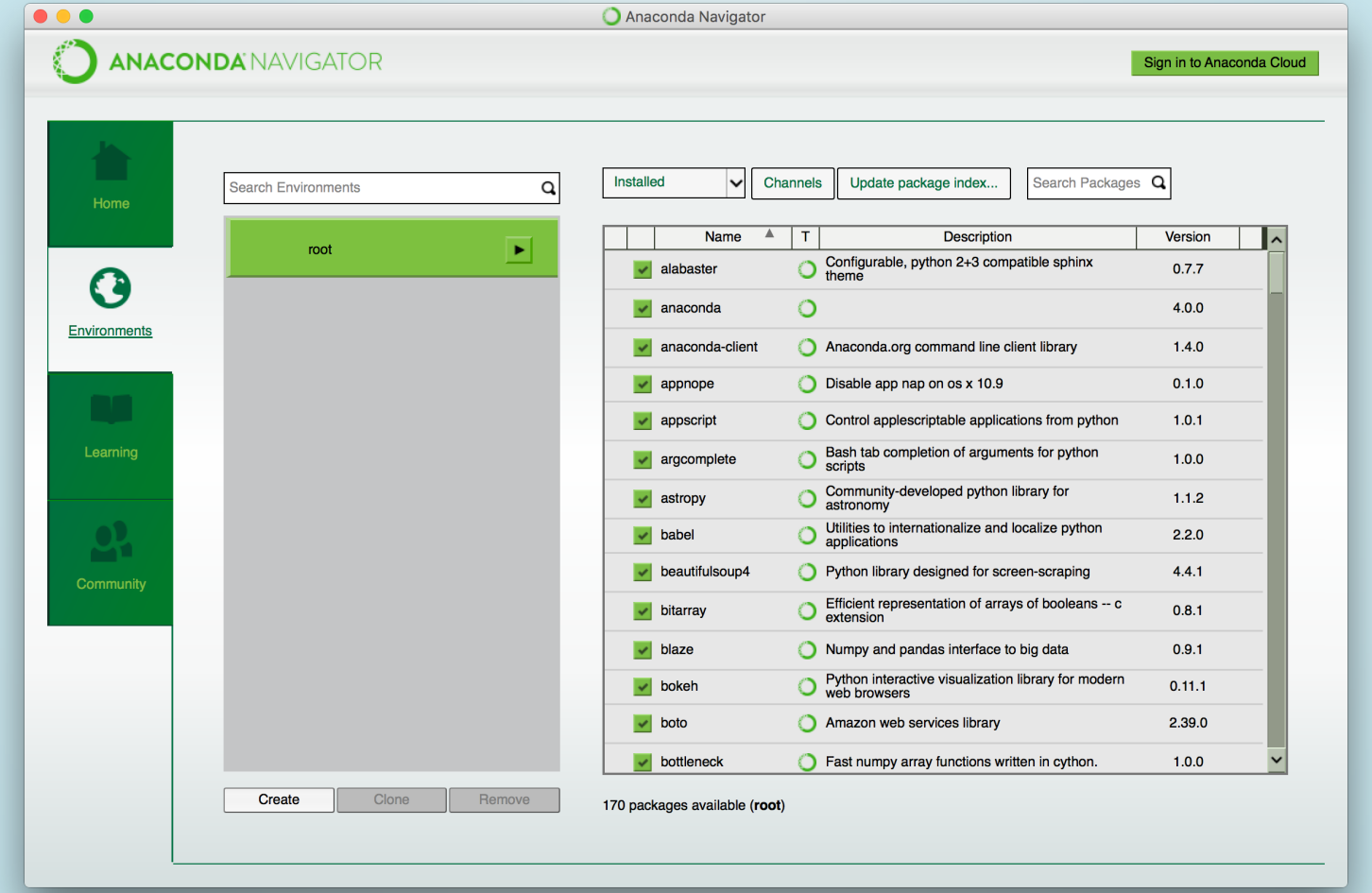
```
# added by Anaconda3 4.0.0 installer
export PATH="/Users/yannick/anaconda/bin:$PATH"
```

- Doing this, calling *python* will use the Python installed by *anaconda* and not the system one.
- Uninstalling *anaconda* is just a mater of removing the *anaconda* directory (and removing the PATH export from the *profile*).

A *navigator* application is also available to launch some application...



… or to install Python packages. But we will see how to do this with the *conda* programme.



THE *CONDA* UTILITY

conda is a Python* package manager.

```
# Search for packages
$ conda search PACKAGE

# Install a package
$ conda install PACKAGE

# Uninstalling a package
$ conda remove PACKAGE

# List installed package
$ conda list
```

* *Not only for Python packages in fact...*

UPDATES

```
# Updating the conda programme (to do once in a while)
$ conda update conda

# Updating the anaconda distribution
$ conda update anaconda

# Updating a specific package
$ conda update PACKAGE
```

Note that *anaconda* is also a package depending on all the standard packages of the distribution, that's why when you “update anaconda” you update the distribution. When you update a specific package, you switch the Anaconda installation to a *custom* version. Subsequently updating the distribution may then downgrade the package.

Keep it simple and only update the distribution.

THE *PIP* UTILITY

pip is the standard tool to install packages from PyPI, the Python Package Index (pypi.python.org).

```
# Search PyPI for packages
$ pip search SOMETHING

# Install a package from PyPI
$ pip install PACKAGE

# Updating a package
$ pip install PACKAGE --upgrade
```

Use *pip* to install packages that are not available with *conda*.

Always prefer *conda* to install a package.

ENVIRONMENTS

Anaconda creates a *root* environment. Separate environments may be created to install a specific set of packages, even with a different Python version.

```
# Creating a new environment to use Python 2.7
$ conda create -n my_py2 python=2.7

# Switching to this environment (look at the new prompt)
$ source activate my_py2
discarding /Users/yannick/anaconda/bin from PATH
prepending /Users/yannick/anaconda/envs/my_py2/bin to PATH
(my_py2)$ _

# From here, all available package, all installations
# are made in the new environment.

# To return to the root environment
(my_py2)$ source deactivate
discarding /Users/yannick/anaconda/envs/my_py2/bin  from PATH
```

When you launch a console, you are always in the *root* environment and have to manually activate the specific environment.

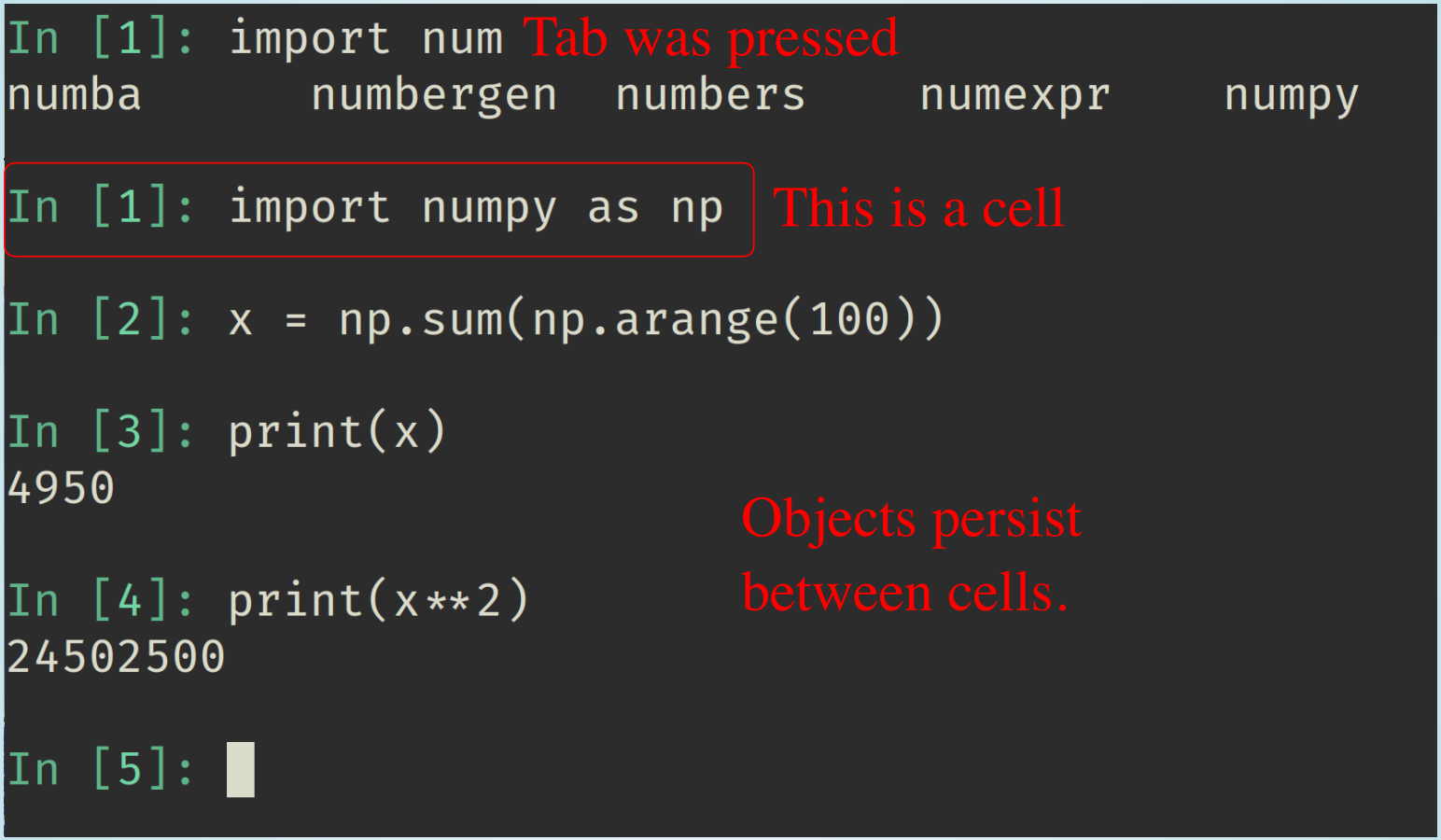
INTERACTIVE PYTHON

If you launch *python* in the console, you can write Python code that is evaluated line per line (it is a REPL - *read, eval, print, loop*). But it's not very user friendly.

IPython was developed to have an interactive Python shell with:

- Code completion (with Tab).
- Access to useful shell commands like *cd* or *ls*.
- Good command history.
- A lot more.

Code is organised in cells (which can by multi-line).



IPython is very handy to perform operations that don't need to be stored in a script.

IPYTHON NOTEBOOK

The notebook was developed to give to IPython an interface similar to Mathematica notebook.

IPython is accessed *via* a web page where one can:

- Write and evaluate Python code.
- Display the results of the code, in particular matplotlib graphics.
- Write some textual content, like a lab notebook.

For instance, the LIGO experiment made a tutorial on the processing of their data:

https://losc.ligo.org/s/events/GW150914/GW150914_tutorial.html

IPYTHON / JUPYTER

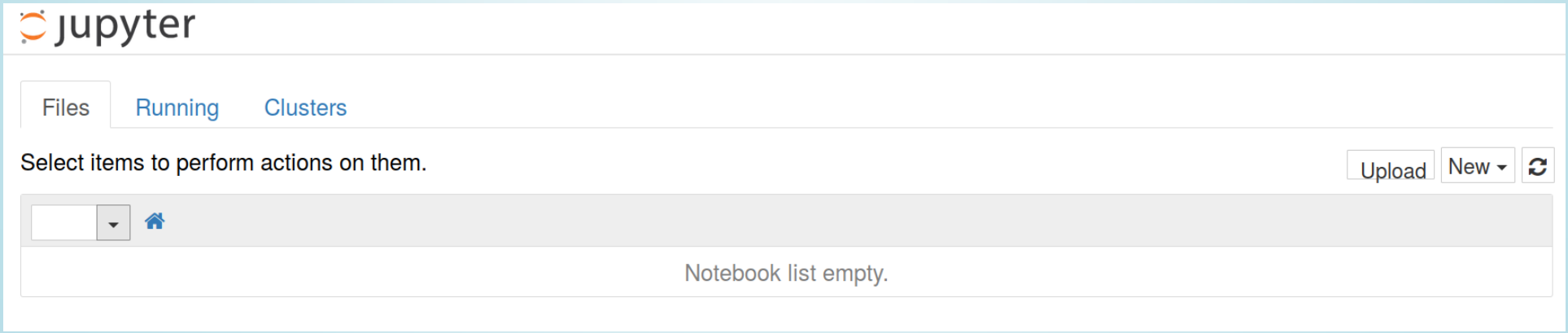
IPython has evolved and can now run code in other languages (*e.g.* GNU-R). Hence, it was renamed to **Jupyter**.

NOTEBOOK - LAUNCHING IT

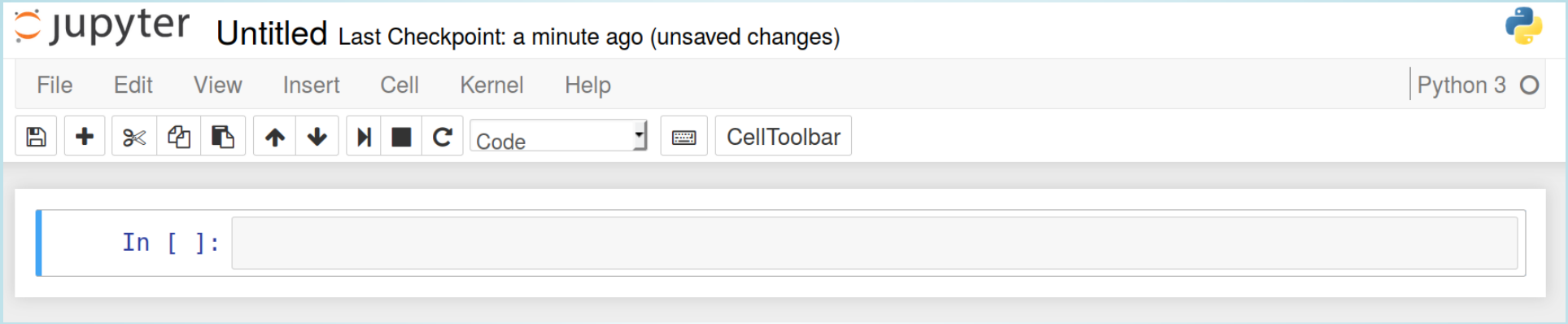
IPython notebook saves the notebooks on disk. We will create a folder where the notebook will be saved on launch the notebook inside.

```
$ mkdir tutorial_notebooks
$ cd tutorial_notebooks
$ jupyter notebook
```

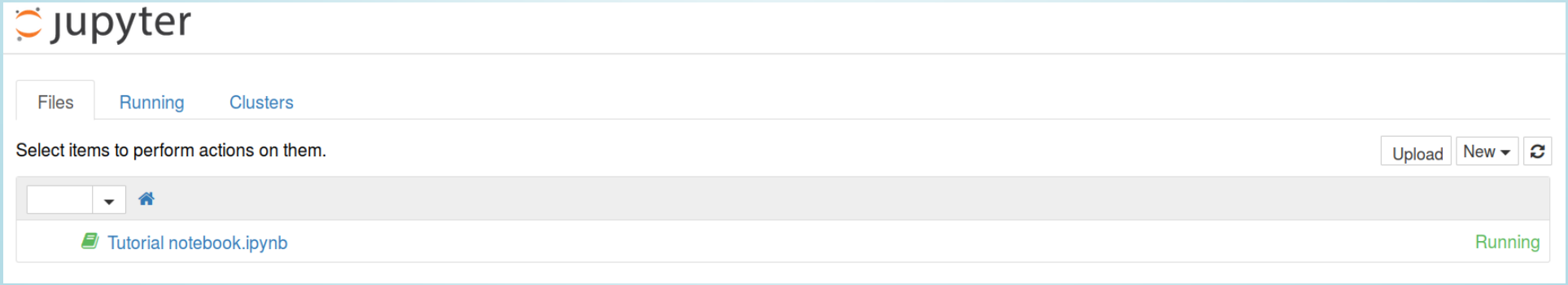
This will spawn a browser showing:



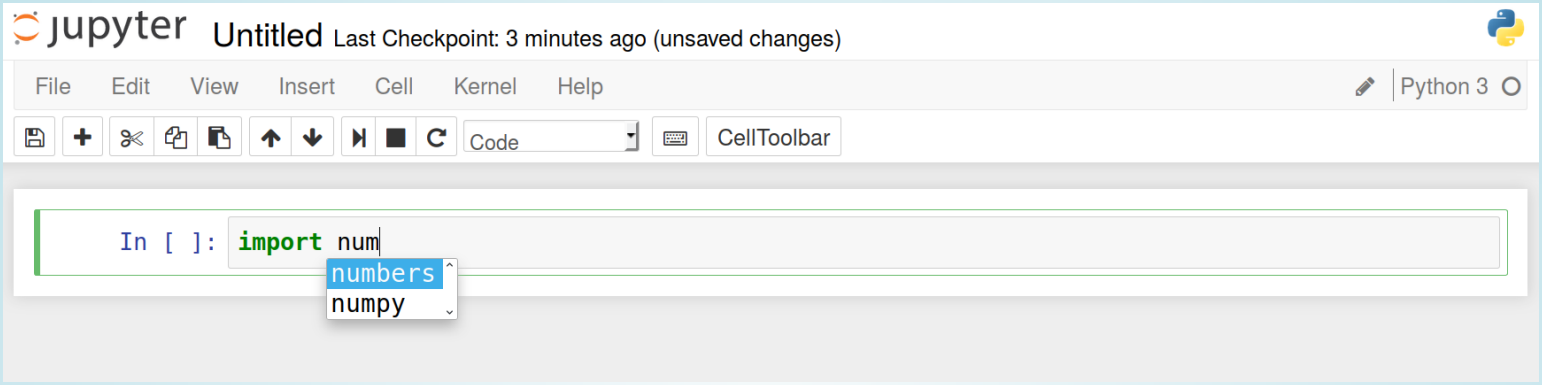
We can create a new notebook doing New → Notebook Python 3. A new browser tab is opened with:



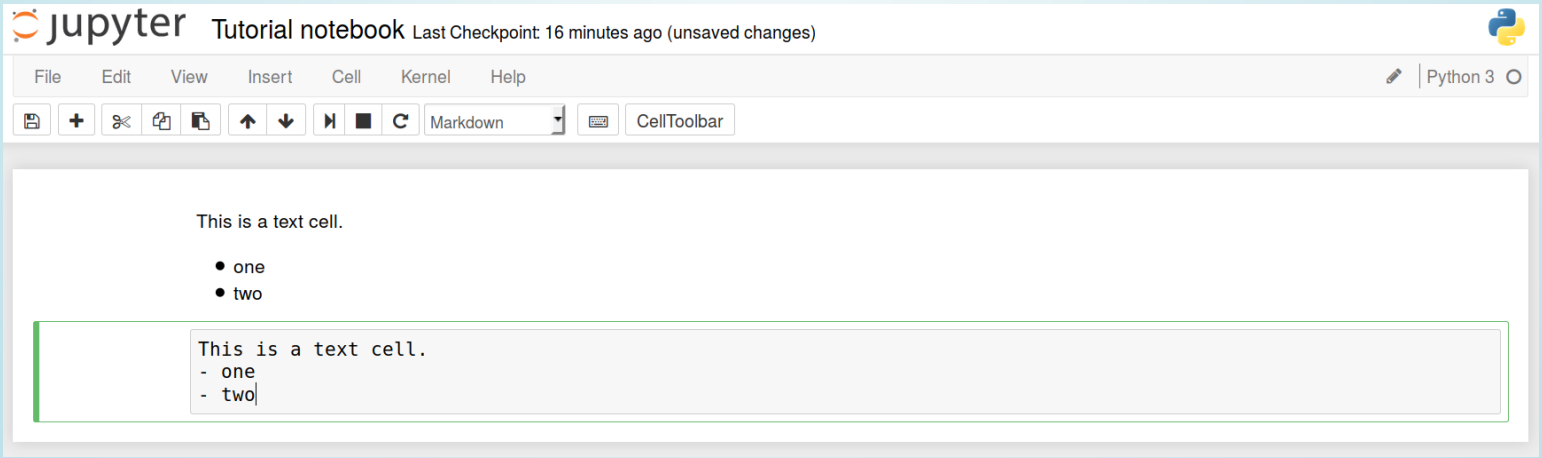
We can rename the notebook with a click on *Untitled*. If we go back to the main tab we can see that the notebook was renamed and that the file in the directory is the notebook name followed by *.ipnb*.



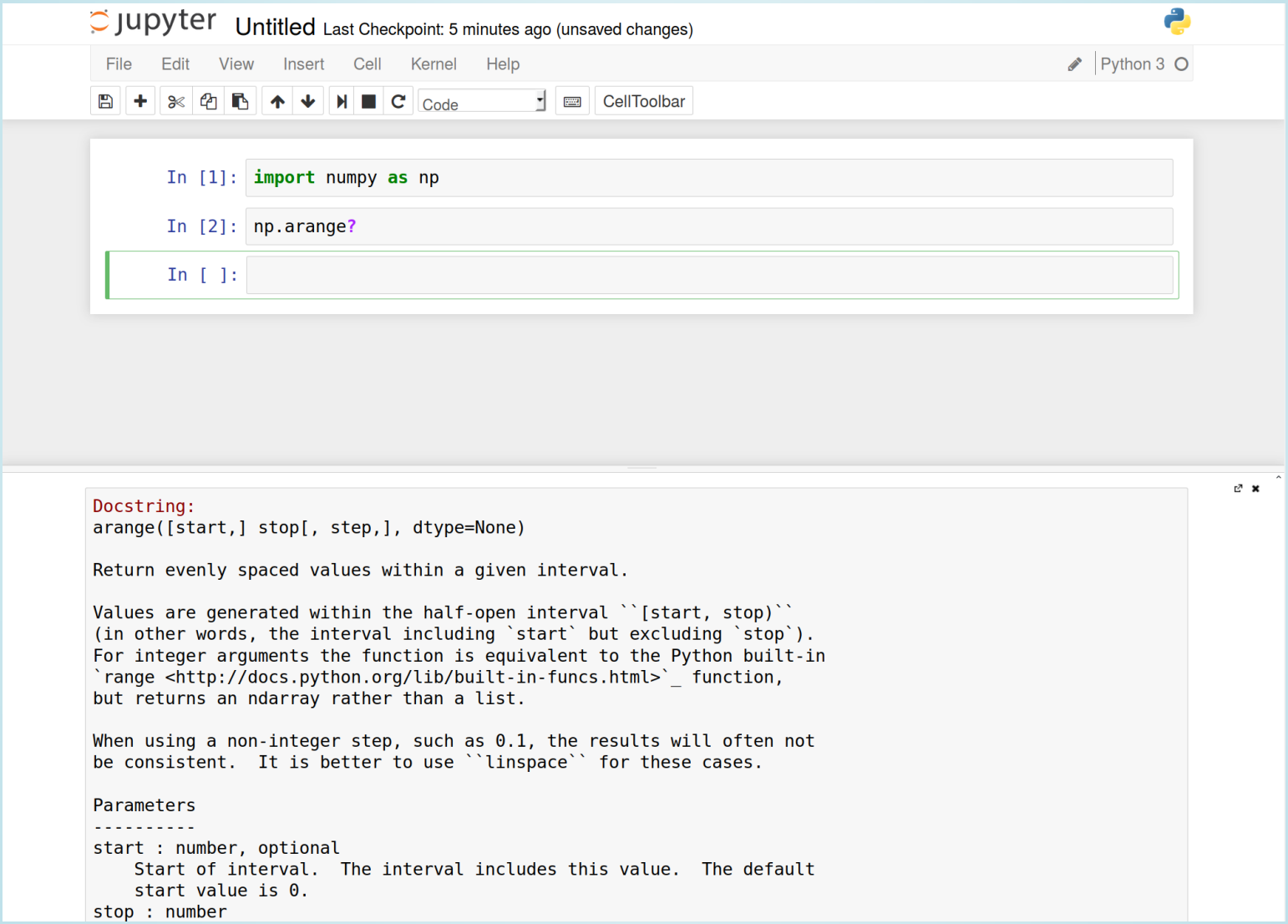
The notebook is organised into cells (like the IPython console). Each cell can contain Python code (with code completion with tab)



or text formatted with *Markdown*.



It's easy to display Python help. You just have to execute a cell with the object you want help on followed by a quotation mark (attached).



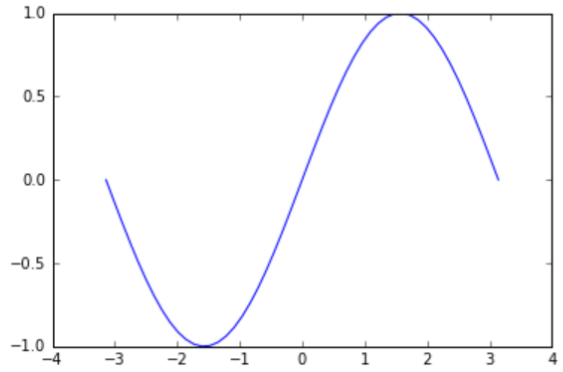
IN-LINE GRAPHICS

You can plot matplotlib figures inside the web page using *%matplotlib inline* at the beginning of the notebook.

```
In [1]: %matplotlib inline
        import numpy as np
        from matplotlib import pyplot as plt

In [2]: x= np.linspace(-np.pi, np.pi)
        plt.plot(x, np.sin(x))

Out[2]: [<matplotlib.lines.Line2D at 0x7f64eb812b38>]
```



Alternatively, you can use *%matplotlib notebook* to have interactive figures (zoom, pan, etc.)

NOTEBOOK CODE EXECUTION

- Each code cell must be executed (*Alt*+*Return* or the play button).
- There is kernel managing a session. Each executed cell modify the kernel environment.
- It's not the order of the cells in the page that define the programme but the order in which the cells are executed.
- There is a shortcut Cell → Run all cells.
- The kernel may be restarted.

TUTORIAL & EXERCISES