An (other one) introduction to Python

Thibaut Lunet

April 1, 2019

What are we talking about

A little bit of history

- Programming language conceived by a Monty Python fan in the 80's
- First released in 1991, then
  - Python 2 in 2000 (community backed development, support $\rightarrow$ 2020)
  - Python 3 in 2008 (major revision)

Current versions (2.7, 3.6) are not entirely compatible, but very close

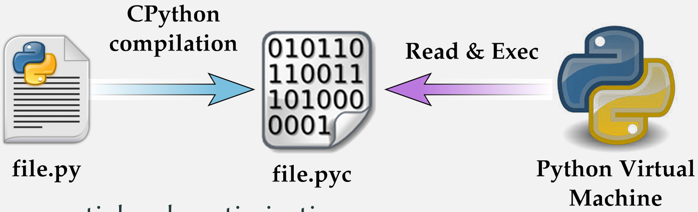
Main motivations

Developers wanted to create a language that would be

- general purpose (they really got that !)
- highly extensible and modular (dynamic language)
- beautiful, simple, easily readable

Functioning principles

Pre-compilation of code files



- Allows partial code optimization
- Totally transparent to user
- Portability on every architecture

Python core (Virtual Machine) is written in C:

- Easy interface with other compiled languages (C/C++, Fortran)
- Same speed as C when reading/writing files

Python vs. Others (Matlab, Fortran, C/C++, ...)

- License-free and open-source ($\neq$ Matlab)
- Huge users community, many (free) packages for many applications
- Extremely easy of use for non-I-love-programming people
(no compilation, no variable declaration, ... $\neq$ Fortran, C/C++)
- Computation can be accelerated using Fortran or C/C++ library ...
- Can scale to very large problems (parallel computing, ...)
- Structured and friendly ways for developing library ($\neq$ Matlab)

Python $=$ excellent solution for algorithm development and prototyping
$\neq$ solution for fast and memory-optimized production codes

Two particular aspects

Implementation is based on indentation

Each code block (condition, loop, function definition, ...) are delimited by indentation, not by brackets or "end ..." commands

- Strange, but also ease implementation
- Forces to write well presented codes

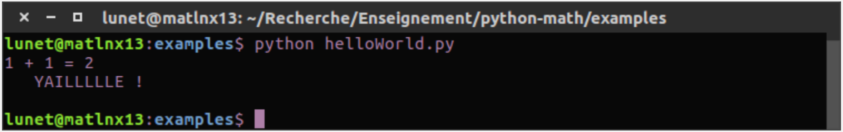
Everything is object

Every programming elements (type, lists, function, ...) are objects, i.e they have their own attributes and methods

- Induces a particular call of functions ("result = object.function(...)")
- Object Oriented programming is natural
- Classical way of calling function ("res = function(...)") still available

Basic use : script and terminal

Code is written in text files, and run using the "python" terminal command

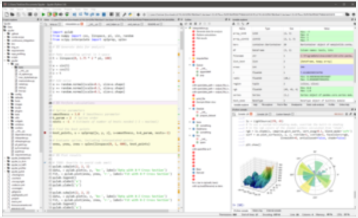


- Need to know how to open a terminal (in Ubuntu, Windows, Mac, ...)
- Line by line command can be executed through the "python" or "ipython" commands

How to use Python

Advanced use : the Spyder Developing Environment

Graphical interface program (∼ Matlab, Mathematica, ...)



- Writing and running scripts interactively
- Dynamical access to variable
- Easy-access documentaion
- Automatic completion
- ...

Spyder demonstration, and hands-on examples

- Installation of Python and Spyder using Anaconda distribution https://www.anaconda.com/distribution $\Rightarrow$ install Python 3.7
- Launch Spyder, and open a script file in the editor
- Test with the following examples, by running the script or typing those commands in the ipython terminal

```python
# Lines starting with # are comments
print('I can do it !')

# Some first examples to run yourself
a = 1.2345
b = 2
print('a+b=', a+b)
```

$\rightarrow$ try to find a and b variables in the variable editor

Basic variables types and operations

```
# Integer
n = 1
m = 7 % 3  # modulo operator, m = 1
k = 7 // 2  # integer division, k = 3
i = int(1.7)  # integer conversion with built-in function, i = 1

# Float: by default, double precision
x = 0.5
y = x/7  # y = 0.07142857142857142
t = float('4.35')  # float conversion, t = 4.35

# Complex
z = 1+1j
w = z + x + n  # Automatic conversion, w = 2.5 + 1j
c = complex(1, 5)  # Other definition, c = 1+5j

# Boolean
p = True
q = (n != 1)*p + (n == 1)*(x < 10)*(y >= 0)  # q = True = 1
r = bool(5)  # Alternative definition, False only for 0 or None
```

→ find how to round any number to the closest integer

Lists

```python
# Definition
l = [1, 2, 5, 6]
# Access elements : l[0]=1, l[2]=5, l[-1]=6, l[-2]=5
# Slice : l[1:3] = [2,5], l[:-3] = [1], l[2:]=[5,6]
l[1] = 4  # Modify second element

# Nested list
nl = [['vive', 'la'], ['saucisse', 2], 'Toulouse']
# Access sublist element : nl[0] = ['vive', 'la']
# Access final element : nl[0][1] = 'la', nl[1][0] = 'saucisse'

# List comprehension
l1 = [i**2 for i in range(10)]
# l1 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
l2 = [3*n + 1 for n in range(10) if n % 2 == 0]
# l2 = [7, 13, 19, 25, 31]

# Built-in functions and methods
length = len(l1)
s = sum(l1)
l2.append(12)
```

→ find all list methods with Spyder autocompletion (.+Tab), and look at their documentation (Ctrl+I)

Strings

Strings are lists with non-mutable elements

```
# Definition
s1 = 'salut'
s2 = 'toi'

# Basic operations (btw, work also for lists)
s3 = s1+' '+s2  # Concatenation
s4 = s1[3:]+s1[:3]  # Slices, s4='utsal'
s5 = s3[::2]  # Extract each two elements, s5='sltti'
s6 = s3[-1::-1]  # Reverse string order, s6='iot tulas'

# Built-in functions and methods
s5 = str(1234)  # Conversion
s6 = s1.upper()  # Change into upper case
s7 = 'BABAORUM'.lower()  # Change into lower case
s8 = 'float : {:1.2f}, int : {:03d}'.format(1.2345, 2)  # String formating
```

→ find more details on format use in https://pyformat.info

→ what your first name looks like in reverse, when you keep only each two letters

Functions

Are delimited by indentation, arguments can have default value

```
def add(a, b=1): # NO NEED to define the type, b has a default value
    return a + b
# add(0.5, 2) = 2.5, add(1) = 2
# add('s', 't')
# add() -> ERROR, add('s') -> ERROR

# Possibility of having a variable number of parameters and outputs
def doSomething(x, y, z, p1=1, p3='red'):
    out1 = p1*(x+y)
    out2 = p3+str(z)
    return out1, out2  # returns a list of two elements

out = doSomething(1, 2, 3)  # out[0] = 3, out[1] = 'red3'
value, flag = doSomething(1, 2, 3)  # value = 3, flag = 'red3'

# Return is not mandatory
def addOneToList(l):
    l = l + [1]
def addOneToList2(l):
    l += [1]  # in-place operation, equivalent to l.append(1)
```

→ what is the difference between **addOneToList** and **addOneToList2** ?

Exercice 1 : the Magic List

```
# Definition of a for loop
for i in range(1, 11):
    print(i)  # print the first integers until 10, starting from 1
```

$\rightarrow$ implement a function **magicList(n,p)**, which returns the last digit of the first **n** integers elevated to the power **p**

Exercice 1 : the Magic List

```
# Definition of a for loop
for i in range(1, 11):
    print(i)  # print the first integers until 10, starting from 1
```

→ implement a function **magicList(n,p)**, which returns the last digit of the first **n** integers elevated to the power **p**

```
# Simple standard definition
def magicList(n,p):
    l = []
    for i in range(1, n+1):
        num = i**p
        numStr = str(num)
        lastDigit = numStr[-1]
        l.append(int(lastDigit))
    return l
```

→ try to implement it in one line

Exercice 1 : the Magic List

```python
# Definition of a for loop
for i in range(1, 11):
    print(i)  # print the first integers until 10, starting from 1
```

→ implement a function **magicList(n,p)**, which returns the last digit of the first **n** integers elevated to the power **p**

```python
# Simple standard definition
def magicList(n,p):
    l = []
    for i in range(1, n+1):
        num = i**p
        numStr = str(num)
        lastDigit = numStr[-1]
        l.append(int(lastDigit))
    return l
```

→ try to implement it in one line

```python
# One-line definition
def magicList(n,p):
    return [int(str(i**p)[-1]) for i in range(1, n+1)]
```

Conditions and loops

```
# If syntax
if 1 == 2:
    print('Tocard')
elif 1 in [0, 3, 4, 5]:  # Not mandatory, with use of a list
    print('Toujours pas')
else:  # Not mandatory
    print("OK d'accord")  # String defined with "" to allow ' character

# For can be applied on a list
for i in [1, 3, 5, 6, 7, 8]:
    print('i = {}'.format(i))
# ... or on two lists (or more)
for i, j in zip([1,2,3], [4,5]):
    print('i={}, j={}'.format(i, j))

# While loop
i = 0
while i < 10:
    print('TAIHOOO-' + str(i))
    i += 1
    if i == 5:
        break  # Allows to escape from the while loop
```

Dictionaries and tuples

Dictionaries are non-ordered lists with keys instead of index

```
# Standard definition
d = {'key1': 1,
     'key2': 'two',
     'key3': {'A': [3, 4, 5],
              'B': True}}

# Access elements : d['key1'] = 1, d['key2'] = 'two'
#                   d['key3'] = {'A': [3, 4, 5], 'B': True}
#                   d['key3']['A'] = [3, 4, 5]
#                   d['key3']['A'][1] = 4
#                   d['key3']['B'] = True

# Built-in function and methods
d1 = dict([('key1',1), ('key2','s')])  # alternative definition
lKeys = d1.keys()  # list of the keys
lValues = d1.values()  # list of the values
```

Tuples are lists of fixed size with non-mutable elements

```
# Standard definition
t = ('salut', 'mon', 'ami')
t[0] = 'bonjour'  # -> ERROR
```

Exercice 2 : the Caesar cipher

$\rightarrow$ decode the following message, encrypted with the Caesar cipher with offset -13 :

"w'nv snvyyv nggraqer"

Exercice 2 : the Caesar cipher

→ decode the following message, encrypted with the Caesar cipher with offset -13 :

"w’nv snvyyv nggraqer"

```
def encodeMsg(msg, offset):
    abc = 'abcdefghijklmnopqrstuvwxyz'
    abc_code = abc[offset:]+abc[:offset]
    # Build a coding dictionary
    dico = {}
    for i in range(len(abc)):
        dico[abc[i]] = abc_code[i]
    # Shorter version
    # dico = dict(zip(abc, abc_code))
    # Translate the message
    msg_code = ''
    for c in msg:
        msg_code += dico[c] if c in dico else c
    # Shorter version
    # msg_code = ''.join([dico[c] if c in dico else c for c in msg])
    return msg_code

print(encodeMsg("w'nv snvyyv nggraqer", 13))
```

The Numpy library

Optimized python library for matrix manipulation

```python
import numpy as np  # Necessary, at the beginning of the script

# Definition
v = np.array([1,2,3])
m = np.array([[1, -1, 0],
              [0, 1, -1],
              [-1, 0, 1]])
# See also np.ones, np.linspace, np.arange, np.eye, ...

# Standard operation
v2 = v*v  # element-wise multiplication
m2 = m*v  # element-wise multiplication for each lines of m
v3 = m.dot(v)  # matrix-vector product

# Particular attributes
s = m.shape
l = m.size

# Element access and slices
a = m[2, 0]
v4 = m[:, 1]
```

Numpy and Scipy

Collection of open source libraries for scientific computing

$\rightarrow$ more details can be found at https://www.scipy.org/about.html

In particular :

- **np.linalg** : Linear Algebra sub-module
- **np.fft** : Fast Fourier Transform sub-module
- **np.random** : Random sampling sub-module

Scipy add a collection of algorithms adapted to numpy arrays, e.g :

```
from scipy import optimize as spo
from scipy import linalg as spl
from scipy import integrate as spig
from scipy import interpolate as spip
from scipy import sparse as sps
```

$\rightarrow$ how to compute the eigenvalues of a matrix ?

$\rightarrow$ how to define a sparse circulant matrix ?

$\rightarrow$ how to compute a curve fitting from given data ?

Plotting data with Matplotlib

Library for data representation similar to Matlab or equivalents

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

x = np.linspace(-1, 1, num=200)
plt.plot(x, np.exp(x), '--', label='Initial data $e^{x}$')
x_mod = x[::20]
y_mod = np.exp(x_mod) + 0.1*np.random.randn(x_mod.size)
plt.plot(x_mod, y_mod, 'o', label='Measured data')

plt.xlabel('$x$')
plt.ylabel('Data')
plt.grid()
plt.legend()
```

→ how to plot data with symbol and lines together ?

→ how to quickly plot the values of a matrix with colorbar ?

Exercise : curve fitting

Perform a curve fitting of the previous measured data, with an objective function of the form $\exp(\alpha x)$

Exercise : curve fitting

Perform a curve fitting of the previous measured data, with an objective function of the form $\exp(\alpha x)$

```
from scipy.optimize import curve_fit
... incoming
```

Incoming ...