

UNE INTRODUCTION À PYTHON

voire un peu plus...

Xavier Olive, ONERA

version 0.9



Python est un langage de programmation :

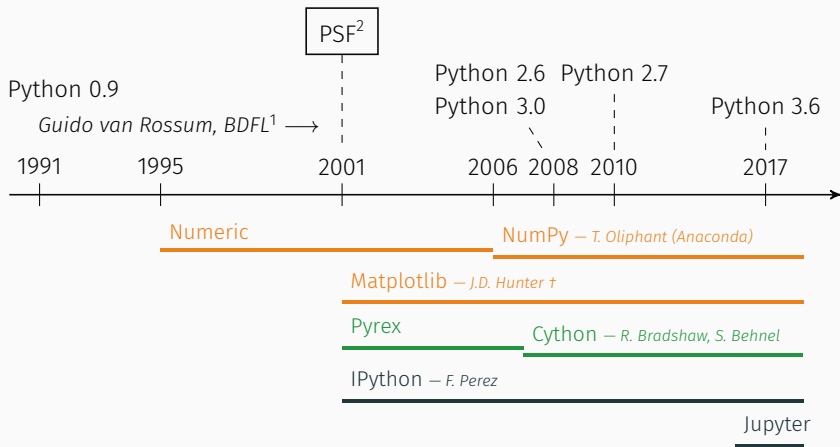
- ▶ interprété ;
- ▶ multi-paradigme ;
- ▶ multi-plateforme ;



utilisé par une communauté importante :

- ▶ une syntaxe simple ;
- ▶ de nombreuses extensions ;
- ▶ libre et gratuit.

HISTORIQUE



¹ BDFL : Benevolent dictator for life

² PSF : Python Software Foundation

Les références à garder à porter de souris :

- ▶ le site officiel : www.python.org;
- ▶ la documentation des *core packages* : docs.python.org;
- ▶ les PEP (Python Enhancement Proposals),
notamment le PEP 8 : *Style Guide for Python Code*;
- ▶ le glossaire ;

et d'une manière générale :

- ▶ la commande `help`;
- ▶ les moteurs de recherche du type www.google.com;
- ▶ les forums du type www.stackoverflow.com.

LES BASES DU LANGAGE

L'INTERPRÉTEUR PYTHON

```
Python 3.4.3 (default, Aug 11 2015, 08:57:25)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> 2 + 3                                # Addition d'entiers
5
>>> 1 + 0.03                             # Entiers et flottants
1.03
>>> (2 + 3) / 7                           # Division flottante
0.7142857142857143
>>> 12 // 5                               # Division entière
2
>>> 0.1 + 0.2                             # Le classique!
0.30000000000000004
>>> print("hello", "y'all")              # Affichage de variables/valeurs
hello y'all
```

OUVREZ VOTRE INTERPRÉTEUR PYTHON!

```
>>> print ("hello")
```

```
hello
```

```
>>> a = 4
```

```
>>> print ("hello, you are now", a)
```

```
hello, you are now 4
```

```
>>> print ("history:", [0, 1, 2, 3])
```

```
history: [0, 1, 2, 3]
```

```
>>> import math
```

```
>>> print (math.pi)
```

```
3.141592653589793
```

```
>>> print ("C-style format: %.2f" % math.pi)
```

```
C-style format: 3.14
```

```
>>> print ("{} has value {}".format("π", math.pi))
```

```
π has value 3.141592653589793
```


- Les variables Python sont des références non typées :

```
>>> a = 12          # a est un entier
>>> a = "hello"     # a est désormais une chaîne de caractères
```

- En revanche, les valeurs Python sont typées :

```
>>> type(2)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type("hello")
<class 'str'>
```

- La variable `_` rappelle la dernière valeur évaluée :

```
>>> _
<class 'str'>
```

- On peut « convertir » le type de certains objets.

```
>>> int(2.4)
2
```

```
>>> str(2)
'2'
```

```
>>> float("3.14")
3.14
```

- On peut appeler ces fonctions sans paramètre :

```
>>> int()
0
```

```
>>> float()
0.0
```

```
>>> str()
''
```

En Python, c'est l'indentation qui définit les blocs;

- PEP 8 recommande l'utilisation de 4 espaces;
(bannir les tabulations)

```
>>> a, b = 15, 9
>>> while (a != b):
...     if (a > b):
...         a = a - b
...     else:
...         b = b - a
...
>>> print ("GCD =", a)

GCD = 3
```

- Python fournit un mécanisme de gestion d'erreurs :

```
>>> float("hello")
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: could not convert string to float: 'hello'
```

```
>>> pi
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

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

- ▶ Le code est écrit dans un fichier avec l'extension `.py`;
- ▶ Modèle d'exécution similaire aux scripts shell :

```
shell> python integrate.py arg 12
```

- ▶ Transformation du script en commande shell :

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

```
shell> ./integrate.py arg 12
```

- ▶ On peut appeler le code d'un autre fichier par un mécanisme de modules :

```
import integrate
```

- ▶ Support par défaut de l'UTF-8 depuis Python 3.

LES STRUCTURES DE DONNÉES

- Les entiers Python sont munis des opérations habituelles.

```
>>> 2 * 2
4
>>> 7 // 3 # integer division
2
>>> 7 % 4 # modulo
3
>>> 7 & 3 # bitwise and (equivalent to "modulo 4")
3
>>> 123 ^ 24 # bitwise xor
99
>>> 2 ** 8 # power
256
>>> 1 << 8 # bit shifting (equivalent to 2 ** 8)
256
```

- On peut manipuler les entiers par leurs représentations binaire, octale, décimale ou hexadécimale.

```
>>> bin(127)
'0b1111111'
>>> oct(127)
'0o177'
>>> hex(127)
'0x7f'
>>> 0b1111111
127
>>> 0o177
127
>>> 0x7f
127
```


- Les entiers Python ont une précision illimitée.

```
>>> 123 ** 24 # no overflow (would need 168 bits!)
143788010446775248848237875203163336494653562343841
```

```
>>> 123 ** 24 * 134 ** 45 # besides it is fast!
754116773913301291674484428969148011550171822575090416487017684
057230784745923800513525439801776494774185798453198912700344174
39450350881010089984
```

Les flottants Python suivent le standard IEEE 754

- ▶ Représentation double précision sur 64 bits :
 - ▶ 1 bit de signe
 - ▶ 11 bits d'exposant (-1022 à 1023)
 - ▶ 52 bits de mantisse (bit 1 implicite)

```
>>> float.hex(1.)
'0x1.0000000000000p+0'
>>> float.hex(2.)
'0x1.0000000000000p+1'
>>> float.hex(.5)
'0x1.0000000000000p-1'
>>> float.hex(10.) # (1 + 4/16) * 2**3
'0x1.4000000000000p+3'
>>> float.hex(1.5) # (1 + 8/16)
'0x1.8000000000000p+0'
```

LES FLOTTANTS

```
>>> float.hex(0.1)
'0x1.999999999999ap-4'
#  $2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + \dots = 0.0999999\dots$ 
>>> float.hex(0.2)
'0x1.999999999999ap-3'
#  $2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + \dots = 0.1999999\dots$ 

>>> float.hex(0.3)
'0x1.333333333333p-2'
#  $2^{-2} + 2^{-5} + 2^{-6} + 2^{-9} + 2^{-10} + \dots = 0.3333333\dots$ 
>>> float.hex(0.1 + 0.2)
'0x1.3333333333334p-2'

>>> 0.1 + 0.2
0.30000000000000004
>>> import sys
>>> 0.1 + 0.2 - 0.3 < sys.float_info.epsilon
True
```

Les complexes sont écrits en notation cartésienne à partir de deux flottants. La partie imaginaire est suffixée par j .

```
>>> a = 3+4j
>>> a.real
3.0
>>> a.imag
4.0
>>> a * a.conjugate()
(25+0j)
>>> abs(a)
5.0
```

Les chaînes de caractères sont écrites à l'aide de guillemets simples, doubles ou retriplées (multi-lignes).

```
>>> "hel" + 'lo'
'hello'
>>> a = """Hello
- dear students;
- dear all"""
>>> a[0]
'H'
>>> a[2:4]
'll'
>>> a[-1]
'l'
>>> a[-8:]
'dear all'
```

Les chaînes de caractères offrent les opérations usuelles.

```
>>> a = "hello"
>>> (a + ' ') * 2
'hello hello '
>>> len(a)
5
>>> a.capitalize()
'Hello'
>>> " hello ".strip()
'hello'
>>> "hello y'all".split()
['hello', "y'all"]

>>> help(str)
```

Le *backslash* est un caractère d'échappement.

```
>>> print ("\u0127") # 16-bit Unicode
```

```
ħ
```

```
>>> print ('hello students\n') # end of line
```

```
hello students
```

```
>>> print (r'hello students\n') # raw string -- ignore backslashes
```

```
hello students\n
```

```
>>> print (u"àççèñtüe") # forcing unicode (Python 2)
```

```
àççèñtüe
```

- Python manipule des chaînes de caractères en UTF-8.
Le type `bytes` donne accès à sa représentation mémoire.

```
>>> type("hello")  
<class 'str'>  
>>> type(b"hello")  
<class 'bytes'>
```

```
>>> b"hello"  
b'hello'  
>>> b"accentué"
```

```
File "<stdin>", line 1
```

```
SyntaxError: bytes can only contain ASCII literal characters.
```

```
>>> "accentué".encode('utf8')  
b'accentu\xc3\xa9'
```

```
>>> [x - b"0"[0] for x in b"123"]  
[1, 2, 3]
```


La liste est un conteneur Python :

- ▶ une séquence de valeurs hétérogènes;
- ▶ tout objet Python peut être placé dans une liste;
- ▶ algorithmique associée (recherche, cycles, etc.)

```
>>> a = [1, "deux", 3.0]
>>> a[0]
1
>>> len(a)
3
>>> a.append(1)
>>> a
[1, 'deux', 3.0, 1]
```

```
>>> a  
[1, 'deux', 3.0, 1]
```

```
>>> a.count(1)  
2
```

```
>>> 3 in a  
True
```

```
>>> a[1] = 2  
>>> a  
[1, 2, 3.0, 1]
```

```
>>> a.sort()  
>>> a  
[1, 1, 2, 3.0]
```

```
>>> a[1:3]  
[1, 2]
```

Python offre des facilités de syntaxe pour les listes.

```
>>> [i for i in range(5)] # similar to list(range(5))
```

```
[0, 1, 2, 3, 4]
```

```
>>> [str(i) for i in range(5)]
```

```
['0', '1', '2', '3', '4']
```

```
>>> [i**2 for i in range(5)]
```

```
[0, 1, 4, 9, 16]
```

```
>>> [i**2 for i in range(5) if i&1 == 0] # smarter than i%2 == 0
```

```
[0, 4, 16]
```

```
>>> [x.upper() for x in "hello"] # even with strings
```

```
['H', 'E', 'L', 'L', 'O']
```

```
>>> a = set()
>>> a.add(1)
>>> a.add(2)
>>> a.add(3)
>>> a.add(1)
>>> a
{1, 2, 3}

>>> a.intersection({2, 3, 4}) # set theory
{2, 3}
>>> a.isdisjoint({4, 5})
True

>>> [i == 2 for i in a] # list comprehension
[False, True, False]

>>> set([1, 2, 4, 1]) # conversion
{1, 2, 4}
```

Les dictionnaires sont des tables associatives conçues pour ordonner des données sur le modèle clé/valeur.

```
>>> d = dict()
>>> d[1] = 'Ain', "Bourg-en-Bresse"
>>> d[2] = 'Aisne', "Laon"
>>> d
{1: ('Ain', 'Bourg-en-Bresse'), 2: ('Aisne', 'Laon')}
>>> d[1]
('Ain', 'Bourg-en-Bresse')
>>> d['01']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: '01'
>>> d[1] = "un"
>>> d
{1: 'un', 2: ('Aisne', 'Laon')}
```

On peut parcourir les clés et les valeurs :

```
>>> d.keys()
dict_keys([1, 2])
>>> d.values()
dict_values([('Ain', 'Bourg-en-Bresse'), ('Aisne', 'Laon')])
>>> d.items()
dict_items([(1, ('Ain', 'Bourg-en-Bresse')), (2, ('Aisne', 'Laon'))])

>>> e = {}
>>> for key, value in d.items():
...     e[("%02d" % key)] = value
>>> e # No order in dictionary!
{'02': ('Aisne', 'Laon'), '01': ('Ain', 'Bourg-en-Bresse')}
```

Le séparateur « virgule » définit le tuple.

```
>>> x = 1, 2, 3
```

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

```
1
```

```
>>> x[0] = -1
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>> a, b, c = x # associativity!
```

```
>>> b
```

```
2
```

```
>>> a, b = x
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: too many values to unpack (expected 2)
```

Une fonction est un bloc de code réutilisable :

- ▶ mots-clés `def` et `return`;
- ▶ arguments, variables locales, variables globales;
- ▶ gestion des cas limites;
- ▶ documentation associée, cas tests (doctest);

Important!

La présentation d'une interface claire, générique et robuste est fondamentale.


```
def gcd(a, b):  
    """Computes the GCD of two positive integers.  
  
    >>> gcd(12, 8)  
    4  
    If necessary, a and b will be cast into integers.  
    >>> gcd(4, 2.3)  
    2  
    """  
    a, b = int(a), int(b)  
    assert (a > 0 and b > 0), "Input values must be positive integers"  
    while a != b:  
        if (a > b):  
            a = a - b  
        else:  
            b = b - a  
    return a
```

La documentation est accessible par la fonction `help`.

```
>>> help(gcd)
```

```
Help on function gcd in module __main__:
```

```
gcd(a, b)
```

```
    Computes the GCD of two positive integers.
```

```
>>> gcd(12, 8)
```

```
4
```

```
If necessary, a and b will be cast into integers.
```

```
>>> gcd(4, 2.3)
```

```
2
```

On peut définir des valeurs par défaut pour les paramètres d'entrée d'une fonction.

```
>>> import cmath
>>> def rotate(value, angle, radian=True):
...     if not radian:
...         angle *= cmath.pi / 180.
...     return value * cmath.exp(angle*1j)
...
>>> rotate(1+2j, cmath.pi/2)
(-2+1.0000000000000002j)
```

En nommant les arguments, l'ordre n'a plus d'importance :

```
>>> rotate(angle=cmath.pi/2, value=1+2j)
(-2+1.0000000000000002j)
```

Le module `doctest` permet d'automatiser les tests unitaires :

```
>>> import doctest
>>> doctest.testmod()
TestResults(failed=0, attempted=2)
```

Il permet également de gérer les exceptions :

```
"""
[...]
>>> gcd(12, -8)
Traceback (most recent call last):
...
AssertionError: Input values must be positive integers
"""
```

Python fournit un mécanisme d'exceptions pour la gestion des cas limites (erreurs et comportements imprévus) :

- ▶ déclenchement par `raise` ;
- ▶ rattrapage par `try/except` ;

Saut de contexte automatique :

- ▶ affichage de la pile d'appel qui a mené à l'erreur.

On peut créer un modèle d'exception ad-hoc mais la sémantique des built-in exceptions suffit généralement.

```
>>> def check(value):  
...     if (int (value) < 0):  
...         raise ValueError("Cannot handle negative value (%s)" % (value))  
...  
>>> check(3)  
>>> check(-7)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 3, in check  
ValueError: Cannot handle negative value (-7)
```

```
>>> def analyze(s):
...     r = []
...     try:
...         vlist = s.split()
...         for v in vlist:
...             r.append(int(v))
...         return r
...     except ValueError as val:
...         print ("Python said: %s" % val)
...     except AttributeError:
...         print ("The argument must be a string of space-separated int")
...
>>> analyze("1 2 3 4 5")
[1, 2, 3, 4, 5]
>>> analyze("1 2 3 4 a")
Python said: invalid literal for int() with base 10: 'a'
>>> analyze(["1 2 3 4 5"])
The argument must be a string of space-separated int
```

Easier to Ask for Forgiveness than Permission (EAFP)

On suppose vraies les conditions nécessaires à l'exécution d'un code pour lever des exceptions si ces hypothèses se révèlent fausses.

≠ Look Before You Leap (LBYP)

```
# LBYP
if "key" in my_dict:
    x = my_dict["key"]
else:
    # handle missing key
```

```
# EAFP
try:
    x = my_dict["key"]
except KeyError:
    # handle missing key
```


- Toutefois, le mécanisme étant coûteux, les exceptions sont à réserver au traitement des cas limites, exceptionnels.

```
>>> text = "We usually consider that pi = 3.14159"
>>> for f in text.split():
...     try:
...         f = float(f)
...         print (f)
...     except ValueError: # not smart in that case...
...         pass
...
3.14159
```

Il faut repenser la logique. On pourrait par exemple écrire :

```
>>> import re # next episode!
>>> for f in re.finditer("\d+(\.\d+)?", text):
...     f = float(f.group())
...     print (f)
...
3.14159
```

Note : `\d+(\.\d+)?` est une *expression régulière* qui décrit le format :
« suite non vide (+) d'entiers (`\d`) suivie éventuellement (?) d'un point (`\.`) et d'une suite non vide d'entiers ».

INTERLUDE



- ▶ Construire et documenter une fonction qui calcule l'aire d'un triangle équilatéral de côté a .
- ▶ Construire et documenter une fonction qui calcule la liste des nombres premiers inférieurs ou égaux à n .
- ▶ Écrire un programme qui lit le fichier dans lequel il est écrit et y compte le nombre d'occurrences de chaque mot.

On pourra partir du modèle suivant :

```
f = open(__file__)  
print (f.readlines())  
f.close()
```

```
import math

def area(x):
    """Computes the area of an equilateral triangle.

    >>> area(2)
    1.7320508075688772
    """
    assert x > 0, "Positive length only"
    return x * x * math.sqrt(3) / 4

if __name__ == '__main__':
    import doctest
    print(doctest.testmod())
```

```
import math

def prime(n):
    """Computes all prime numbers below n.

    Computes the sieve of Eratosthenes
    >>> prime(20)
    {1, 2, 3, 5, 7, 11, 13, 17, 19}
    """
    assert n > 0, "Positive integers only"
    p = set(range(1, n))
    for i in range(2, int(math.sqrt(n)) + 1):
        p = p - set(x*i for x in range(2, n//i + 1))
    return p

if __name__ == '__main__':
    import doctest
    print(doctest.testmod())
```

```
import string


f = open(__file__)
d = {}
for x in f.readlines():
    for s in string.punctuation: # !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
        x = x.replace(s, " ")
    for p in x.split():
        if p in d.keys():
            d[p] += 1
        else:
            d[p] = 1

for key in sorted(d.keys()):
    print("{}: {}".format(key, d[key]))

f.close()
```

LA BIBLIOTHÈQUE STANDARD PYTHON

Python propose par défaut :

- ▶ des fonctions intégrées (comme `print`, `help`, etc.);
- ▶ des bibliothèques d'utilitaires usuels;
 *batteries included*

<https://docs.python.org/3.4/library/index.html>

6.2. `re` – Regular expression operations

8.1. `datetime` – Basic date and time types

9.2. `math` – Mathematical functions

16.1. `os` – Miscellaneous operating system interfaces

26.2. `doctest` – Test interactive Python examples

29.1. `sys` – System-specific parameters and functions

Une expression régulière décrit un motif applicable à une chaîne de caractères.

Syntaxe de base :

- ▶ les caractères usuels, `a` ou `0`, se décrivent eux-mêmes;
- ▶ `.` décrit un caractère quelconque;
- ▶ `*` marque 0 occurrence ou plus d'un motif;
- ▶ `+` marque 1 occurrence ou plus d'un motif;
- ▶ `?` marque 0 ou 1 occurrence d'un motif;
- ▶ `[]` décrit un ensemble de caractères;
- ▶ `()` regroupe un motif (à identifier, répéter, etc.)
- ▶ des raccourcis : `\d` pour `[0-9]`, `\w` pour `[a-zA-Z0-9_]` en ASCII, etc.

```
>>> import re

# Match simple words
>>> re.search("and", "lorem ipsum dolor sit amet")
>>> re.search("or", "lorem ipsum dolor sit amet")
<_sre.SRE_Match object; span=(1, 3), match='or'>
>>> re.findall("or", "lorem ipsum dolor sit amet")
['or', 'or']

# Match hexadecimal colors
>>> re.search(r"([\da-fA-F]){6}", "color=#aa3d1f;")
<_sre.SRE_Match object; span=(7, 13), match='aa3d1f'>
>>> _.group()
'aa3d1f'
```

```

>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)          # The entire match
'Isaac Newton'
>>> m.group(1)          # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)          # The second parenthesized subgroup.
'Newton'

# Find all adverbs
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly

```

```
>>> import datetime as dt

>>> now = dt.datetime.now()
>>> now
datetime.datetime(2016, 5, 19, 9, 40, 30, 390247)
>>> print (now)
2016-05-19 09:40:30.390247

>>> now + dt.timedelta(minutes=10)
datetime.datetime(2016, 5, 19, 9, 50, 30, 390247)

>>> xmas = dt.datetime(now.year, 12, 25)
>>> xmas - now
datetime.timedelta(219, 51569, 609753)
>>> print (xmas - now)
219 days, 14:19:29.609753

>>> dt.datetime.utcnow()
datetime.datetime(2016, 5, 19, 7, 42, 11, 792561)
```

```
>>> import os
>>> import os.path
>>> import sys

>>> sys.platform
'darwin'

>>> help(sys.argv) # you may prefer argparse

>>> os.curdir, os.path.abspath(os.curdir)
('.', '/Users/xo')

>>> [x for x in os.listdir() if re.match(".*rc", x)]
['.bashrc', '.curlrc', '.vimrc', '.wgetrc', '.zshrc']
```

- ▶ Python est livré avec des fonctions de bases, notamment `print`, `type` ou `help`;
- ▶ D'autres fonctions s'appliquent à des structures qui répondent à des spécifications particulières :
 - ▶ `len` renvoie la longueur d'une structure séquentielle;
 - ▶ `max` renvoie le plus grand élément d'une structure séquentielle si ces éléments peuvent se comparer entre eux;
 - ▶ `sum` renvoie la somme d'éléments d'une structure séquentielle si ces éléments peuvent se sommer entre eux;
 - ▶ `sorted` renvoie une liste triée d'éléments d'une structure séquentielle si ces éléments peuvent se comparer entre eux;

La fonction `range` produit une forme d'itération :

- ▶ les éléments sont générés explicitement, un par un, à chaque passage dans une boucle ;
- ▶ `len`, `sorted`, etc. savent manipuler cette structure *itérable*.

```
>>> r = range(1, 1024, 3) # range does not produce a list
>>> r
range(1, 1024, 3)

>>> import collections
>>> isinstance(r, collections.Iterable)
True
>>> list(r)
# (truncated: the full list is expanded in memory)
```


enumerate et zip fonctionnent de la même manière :

```
# enumerate
>>> s = [x**2 for x in range(5)]
>>> s
[0, 1, 4, 9, 16]
>>> list(enumerate(s))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]
```

```
# zip
>>> x = [1, 2, 7, 4]
>>> y = [2*t+1 for t in x]
>>> list(zip(x, y))
[(1, 3), (2, 5), (7, 15), (4, 9)]
```

- Le mot-clé `yield` remplace le mot `return` :

```
>>> def syracuse(n):  
...     yield n  
...     while n != 1:  
...         if n % 2 == 0:  
...             n = n // 2  
...         else:  
...             n = 3*n + 1  
...         yield n  
...  
>>> syracuse(58)  
<generator object syracuse at 0x7f9386164d38>  
>>> list(syracuse(58)) # or for s in syracuse(58): ...  
[58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

- Des mots-clés venus du paradigme *fonctionnel*.

```
>>> temperatures = [36.5, 37, 37.5, 38, 39] # celsius
>>> fahrenheit = map(lambda t: 9/5 * t + 32, temperatures)
>>> list(fahrenheit)
[97.7, 98.60000000000001, 99.5, 100.4, 102.2]
```

```
>>> fever = filter(lambda t: t > 37.5, temperatures)
>>> list(fever)
[38, 39]
```

- La compréhension de liste peut être plus lisible.

```
>>> [9./5 * t + 32 for t in temperatures]
[97.7, 98.60000000000001, 99.5, 100.4, 102.2]
>>> [t for t in temperatures if t > 37.5]
[38, 39]
```

Comment appliquer à une liste $[x_0, x_1 \dots x_n]$ l'opération :

$$f(\dots f(f(x_0, x_1), x_2) \dots x_n)$$

```
>>> import functools
>>> l = [1, 7, 2, 4]
>>> functools.reduce(lambda x, y: x + y, l) # prefer sum here
14
>>> functools.reduce(lambda x, y: x * y, l)
56
>>> functools.reduce(lambda x, y: 10*x + y, l)
1724
```

INTERLUDE



- ▶ Construire et documenter une fonction qui lève une exception `ValueError` si une chaîne de caractères ne représente pas un numéro d'immatriculation de véhicule.
- ▶ Lister les éléments du dossier `/tmp` qui ont été modifiés il y a moins de 24 heures.
- ▶ Construire et documenter une fonction qui décompose une chaîne de caractères représentant un entier pour reconstruire l'entier correspondant.

👉 *ne pas utiliser `int()`*

```
import re

def verify(string):
    """Match valid registration numbers.

    >>> verify("AA-229-BY")
    >>> verify("1234-WC-75")
    Traceback (most recent call last):
        ...
    ValueError: Not a valid registration number: 1234-WC-75
    """
    if not re.match(r"[A-Z]{2}-\d{3}-[A-Z]{2}$", string):
        raise ValueError("Not a valid registration number: %s" % string)

if __name__ == "__main__":
    import doctest
    print(doctest.testmod())
```

```
import os
import os.path
import datetime as dt

files = []
for f in os.listdir("/tmp"):
    timestamp = os.path.getmtime(os.path.join("/tmp", f))
    ts_date = dt.datetime.fromtimestamp(timestamp)
    delta_max = dt.timedelta(days=1)
    if dt.datetime.now() - ts_date < delta_max:
        files.append(f)

print(files)
```



```

import functools

def horner(string):
    """Horner's method. Equivalent to int(string) for strings.

    >>> horner("12583")
    12583
    >>> horner("1234a")
    Traceback (most recent call last):
        ...
    TypeError: 'a' is not a number
    """
    tab = [x - b"0"[0] for x in string.encode()]
    for i, x in enumerate(tab):
        if x > 9 or x < 0:
            raise TypeError("'%s' is not a number" % string[i])
    return functools.reduce(lambda rec, x: rec * 10 + x, tab, 0)

```

PYTHON SCIENTIFIQUE

Un éventail de bibliothèques Python performantes fait autorité dans la communauté Python *scientifique* :

- ▶ **NumPy** joue sur la performance des traitements numériques basés sur les tableaux;
- ▶ **SciPy** intègre des algorithmes scientifiques (optimisation, algèbre linéaire, interpolation, intégration, etc.);
- ▶ **Matplotlib** permet des tracés graphiques à la Gnuplot;
- ▶ **IPython** propose une version plus interactive de l'interpréteur Python habituel.

- ▶ NumPy fournit le type `ndarray` ;
- ▶ Les tableaux sont constitués d'éléments du même type ;
- ▶ Les opérations arithmétiques sont vectorisées/optimisées.

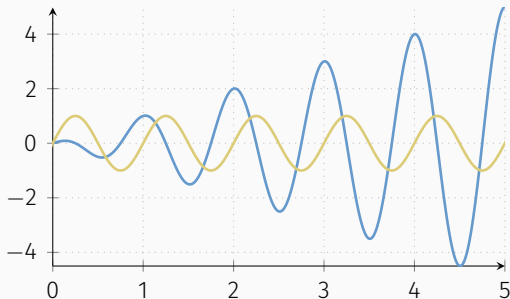
```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4]) # <class 'numpy.ndarray'>
>>> a.dtype
dtype('int64')
>>> a
array([1, 2, 3, 4])
>>> b = 2.5 * a # scalar multiplication
>>> b.dtype
dtype('float64')
>>> b - a # vector subtraction
array([ 1.5,  3. ,  4.5,  6. ])
```

- NumPy permet d'exprimer des traitements de manière efficace et élégante ;

Exemple de calcul des décimales de π (Monte-Carlo) :

```
>>> import numpy as np
>>> size = 1e8
>>> positions = np.random.rand(size, 2)
>>> x, y = positions[:, 0], positions[:, 1] # memory views
>>> x0, y0 = 0.5, 0.5
>>> distances = (x - x0)**2 + (y - y0)**2
>>> len(np.where(distances < .25)[0])/size * 4
3.14154144
```

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 5, 1000)
>>> plt.plot(x, x * np.cos(2*np.pi*x))
>>> plt.plot(x, np.sin(2*np.pi*x))
>>> plt.show()
```



IPython propose un interpréteur plus puissant que celui par défaut, avec notamment :

- ▶ l'auto-complétion ;
- ▶ la coloration syntaxique ;
- ▶ l'aide *docstring* accessible via le préfixe ? ou ?? ;
- ▶ des *commandes magiques* accessibles via le préfixe % :
 - ▶ `%timeit` évalue le temps d'exécution de la commande qui suit ;
- ▶ l'exécution de commandes systèmes préfixées par ! :
 - ▶ `!ls` (Unix ou Mac) liste les fichiers du répertoire courant.

```
Python 3.4.3 (default, Aug 11 2015, 08:57:25)
Type "copyright", "credits" or "license" for more information.
```

```
IPython 4.1.2 -- An enhanced Interactive Python.
```

```
?          -> Introduction and overview of IPython's features.
```

```
%quickref -> Quick reference.
```

```
help       -> Python's own help system.
```

```
object?    -> Details about 'object', use 'object??' for extra details.
```

```
In [1]: import numpy as np
```

```
In [2]: np.si<TAB>
```

np.sign	np.sin	np.singlecomplex
np.signbit	np.sinc	np.sinh
np.signedinteger	np.single	np.size

```
In [2]: np.si
```


Le format « Jupyter notebook » (extension `.ipynb`) permet :

- ▶ d'écrire du texte dans un langage à balises simple ;
- ▶ d'exécuter du code dans un langage interprété (IPython) ;
- ▶ d'exploiter, de présenter et d'analyser les résultats dans le même document.

À retenir!

La commande `%matplotlib inline` permet l'inclusion de figures produites par Matplotlib au sein du même document.

INTERLUDE



- ▶ Le notebook `01-scientific-python.ipynb` contient une présentation interactive des modules NumPy et Matplotlib.
- ▶ Le notebook `02-pandas-dataframe.ipynb` propose une introduction à Pandas, surcouche à Numpy qui plaît aux utilisateurs de R ou de Microsoft Excel.
- ▶ Le notebook `03-scipy-integrate.ipynb`, présente les fonctions d'intégration proposées par SciPy.
- ▶ Le notebook `04-scipy-optimize.ipynb` montre un exemple d'application des outils d'optimisation de SciPy.

MODULES – INTERFACES

Le module est l'unité de nommage Python (\approx fichier).
C'est une **unité de services** qui fournit :

- ▶ des constantes;
- ▶ des types;
- ▶ des fonctions.

```
import numpy
import matplotlib.pyplot as plt
from math import pi, cos
```

La commande `import` procède à de l'interprétation de code Python et/ou au chargement de bibliothèques compilées.

L'interpréteur recherche le module dans :

- ▶ le répertoire courant;
- ▶ la liste de répertoires du `PYTHONPATH`;
- ▶ les répertoires systèmes.

Important!

Chaque module n'est importé **qu'une seule fois** par session. Si le module est changé, il faut redémarrer l'interpréteur.

Note : Python produit et maintient une version compilée d'un fichier à l'extension `.pyc` dans un dossier `__pycache__`.

Les modules présentent une interface à des *services génériques et réutilisables*, tout en masquant les détails techniques : algorithmique, code, choix du langage !

Rappel

La présentation d'une interface claire, générique et robuste est fondamentale.

Questions fondamentales :

- ▶ Quelles sont les fonctions à forte plus-value ?
- ▶ Où trouver la documentation et des exemples d'utilisation ?

Python fournit des outils performants pour la livraison de modules, mais il faut penser en amont à :

- ▶ une méthodologie (tests, documentation, développement);
- ▶ des bonnes pratiques sur l'interface.

On distingue souvent :

- ▶ tests unitaires (non régression, à granularité fine);
- ▶ tests d'intégration (l'articulation du code);
- ▶ tests de performance.

Python propose `distutils` mais la Python Packaging Authority maintient une alternative plus complète, `setuptools`.

L'arborescence suivante est courante :

```
geography.git/  
- geography/  
  - __init__.py  
  - core.py  
  - geodesy.py  
  - projection.py  
- tests/  
- license.txt  
- readme.txt  
- setup.py
```

- ▶ Le fichier `__init__.py` permet de charger des fichiers dans un répertoire. Il est souvent vide ;
- ▶ Les `import` relatifs au module courant sont permis :

```
import .core  
from ..foo import bar # go up one directory
```

- ▶ Le fichier `setup.py` décrit le module ;
- ▶ Construction, installation, distribution du *paquet* :

```
python setup.py install # build and install  
python setup.py install --prefix=/my/favourite/directory  
python setup.py bdist_wheel # produce a distributable archive
```

```
from setuptools import setup

setup(name="geography",
      version="0.1",
      author="Xavier Olive,",
      author_email="me@example.com",
      description="A fantastic module for geography",
      long_description=open("readme.txt").read(),
      license="MIT",
      packages=["geography", ], # folder name
      install_requires=["numpy>=1.4", ],
      )
```

- ▶ Le site <https://pypi.python.org/pypi>³ recense des milliers de paquets.
- ▶ **pip**⁴ est un outil de gestion des paquets et dépendances :

```
pip install numpy
pip uninstall numpy
pip install numpy=1.9
pip install -r requirements.txt
```
- ▶ Notion d'environnement virtuel.
 - ▶ le paquet **virtualenv** ;
 - ▶ les environnements virtuels dans Anaconda.

³ The Python Package Index

⁴ Acronyme récursif pour « Pip Installs Packages »

INTERLUDE



- ▶ Reprendre le code des séances précédentes et commencer la production d'un module livrable.

PYTHON ORIENTÉ OBJET

Les valeurs, ou **instances**, Python ont un type, ou **classe** :
`int`, `bool`, `str`, `list` ou `numpy.ndarray`, etc.

Chaque classe présente des propriétés :

- ▶ on peut comparer des `int`, des `float` :

```
>>> 1 < 3.14  
True
```

- ▶ on peut parcourir et indexer des `str`, des `list` :

```
>>> 'object'[2]  
'j'
```

- ▶ certaines classes exposent des **méthodes** :

```
>>> 'help!'.upper()  
'HELP!'
```


Les fonctions Python sont codées en supposant que les données d'entrées ont les mêmes propriétés :

- `sorted` trie une structure **séquentielle** formée de valeurs **comparables**.

```
>>> sorted([1, 4, 7])  
[1, 4, 7]  
>>> sorted("help")  
['e', 'h', 'l', 'p']
```

Le typage canard (*duck-typing*)

« *S'il marche comme un canard et cancanne comme un canard, alors c'est un canard !* »

- ▶ Python ne s'intéresse pas tant aux objets qu'à leur comportement.
- ▶ Si deux instances offrent la même interface, alors on peut appeler les mêmes fonctions dessus.
- ▶ Le contrôle est assuré par les exceptions.

👉 style EAFP

L'orienté objet est un style de programmation qui repose sur trois grands principes :

- ▶ la notion d'**encapsulation** : un objet embarque des propriétés (attributs, méthodes, etc.);
- ▶ la notion d'**interface** : un objet présente et documente des services tout en cachant sa structure interne;
- ▶ la notion de **factorisation** : on met en commun des objets au comportement similaire.

Python⁵ offre des facilités pour mettre en œuvre ces principes.

⁵ et a fortiori les langages orienté objet

On souhaite trier une liste de points (coordonnées complexes) dans le plan par module croissant :

```
>>> coords = [1+2j, 3j, 2+1j, 4]
>>> sorted(coords)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: complex() < complex()
```

- *Il n'y a pas de relation d'ordre total sur les complexes.*

On crée un objet `Point` qui hérite des complexes :

```
>>> class Point(complex):
...     """A class for representing 2-dimension points."""
...     def __lt__(self, pt):
...         """An order relation based on the L2-norm."""
...         return self.real**2 + self.imag**2 < pt.real**2 + pt.imag**2
...
>>> points = [Point(p) for p in coords]
>>> sorted(points)
[(1+2j), (2+1j), 3j, (4+0j)]
```

Le `Point` garde toutes les propriétés des complexes.

```
>>> points[0] * points[1]
```

```
5j
```

```
>>> 2+3j + Point(3j)
```

```
(2+6j)
```

```
>>> points[1].real
```

```
2.0
```

Les `doctests` produisent la documentation habituelle.

```
>>> help(Point)
```

Une instance Python permet l'ajout dynamique d'attributs.

```
class Vector(object):  
    pass
```

```
v = Vector()  
v.x = 4.2  
v.y = 2.3  
v.z = 0.2
```

```
print (v.x, v.y, v.z)
```

similaire à un dictionnaire...

```
v = {'x': 4.2, 'y': 2.3, 'z': 0.2}
```

```
print (v.x, v.y, v.z)
```

CONSTRUCTION D'UNE CLASSE

```
import numpy as np

class Vector(object):
    # By convention, self refers to the instance itself
    def norm(self):
        """The L2-norm of the structure."""
        return np.sqrt(self.x**2 + self.y**2 + self.z**2)

v = Vector()
v.x = 4.2
v.y = 2.3
v.z = 0.2

print (v.norm()) # 4.792702786528704
```


CONSTRUCTION D'UNE CLASSE

```
class Vector(object):
    null = 0.0
    def norm(self):
        """The L2-norm of the structure."""
        return np.sqrt(self.x**2 + self.y**2 + self.z**2)
    # Methods may modify the instance.
    def zero(self):
        """Bring back to zero."""
        self.x, self.y, self.z = self.null, self.null, self.null

v = Vector()
v.x = 4.2
v.y = 2.3
v.z = 0.2
v.zero()

print (v.norm()) # 0.0
```

CONSTRUCTION D'UNE CLASSE

```
class Vector(object):
    null = 0.0
    def __init__(self, x, y, z):
        """Creates a new vector from its coordinates."""
        self.x, self.y, self.z = x, y, z
    def norm(self):
        """The L2-norm of the structure."""
        return np.sqrt(self.x**2 + self.y**2 + self.z**2)
    def zero(self):
        """Bring back to zero."""
        self.x, self.y, self.z = self.null, self.null, self.null

v = Vector(4.2, 2.3, 0.2)
print (v.norm()) # 4.792702786528704

v.zero()
print (v.norm()) # 0.0
```

☞ On n'a jamais dit que x , y et z doivent être des flottants.

Ils doivent simplement pouvoir :

- ▶ être additionnés;
- ▶ être valides vis-à-vis du calcul de la norme.

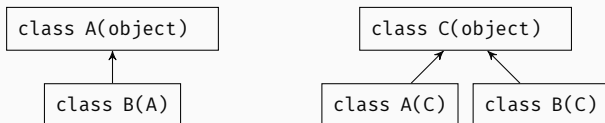
```
>>> a = np.linspace(0, 5, 3)
>>> b = np.linspace(5, 0, 3)
>>> c = 0
>>> v = Vector(a, b, c)
>>> v.norm()
array([ 5.          ,  3.53553391,  5.          ])
```

- ▶ Il ne s'agit pas de décrire le monde.
- ▶ La factorisation se conçoit sur les valeurs (les données). Elle se formalise ensuite sur les types.
- ▶ La factorisation est arbitraire.
- ▶ La factorisation dépend de l'application cible.

On peut factoriser deux classes A et B si :

- ▶ elles ont la même interface ;
- ▶ leur interface a une partie commune.

La factorisation peut s'exprimer suivant ces modèles :



FACTORISATION – EXEMPLE

```
class ThermalPowerStation(object):
    unitcost = 123 # dollar per MWh
    def production(self, period):
        """Computes the production during a given period of time (MWh)."""
        # specific code
    def pollution(self, period):
        # specific code
    def cost(self, period):
        """Computes the cost of production during a given period of time."""
        return self.unitcost * self.production(period)

class WindTurbine(object):
    unitcost = 12 # dollar per MWh
    def production(self, period):
        """Computes the production during a given period of time (MWh)."""
        # specific code
    def cost(self, period):
        """Computes the cost of production during a given period of time."""
        return self.unitcost * self.production(period)
```

FACTORISATION – EXEMPLE

```
class PowerStation(object):
    def pollution(self, period): raise NotImplementedError()
    def cost(self, period):
        """Computes the cost of production during a given period of time."""
        return self.unitcost * self.production(period)

class ThermalPowerStation(PowerStation):
    unitcost = 123 # dollar per MWh
    def production(self, period):
        # specific code
    def pollution(self, period):
        # specific code

class WindTurbine(PowerStation):
    unitcost = 12 # dollar per MWh
    def production(self, period):
        # specific code
```

INTERLUDE



- ▶ Construire et documenter une classe `Container` qui contient un `set` de valeurs accompagné d'un ensemble de valeurs autorisées :
 - ▶ la méthode `add` ajoute une valeur si elle est autorisée ;
 - ▶ la méthode `extend` ajoute une valeur à la liste autorisée ;
 - ▶ la méthode `restrict` enlève une valeur de la liste des valeurs autorisées.

Note : Il faut aussi gérer et documenter les cas limites.

```
"""Specifications and unit tests for the exercise.
```

```
>>> e = Container()
>>> for v in [1, 2, 'a', 3.14]:
...     e.extend(v)
>>> e.add(3.14)
>>> e.add(2)
>>> e.add(2)
>>> e
Container({2, 3.14})

>>> e.add(5)
Traceback (most recent call last):
...
ValueError: Value '5' is not allowed.
```

```
>>> e.extend(5)
>>> e.add(5)

>>> e.restrict(1)
>>> e.restrict(2)
Traceback (most recent call last):
...
ValueError: Value '2' is present in the Container.
"""
```

```
class Container(set):
    allowedValues = set()
    def extend(self, v):
        """Add a value in the list of allowed values."""
        self.allowedValues.add(v)
    def add(self, v):
        """Add a value in the set."""
        if v not in self.allowedValues:
            raise ValueError("Value '%s' is not allowed." % v)
        set.add(self, v)
    def restrict(self, v):
        """Remove a value from the list of allowed values."""
        if v in self:
            raise ValueError("Value '%s' is present in the Container." % v)
        self.allowedValues.remove(v)


if __name__ == '__main__':
    import doctest
    print(doctest.testmod())
```

REPRENONS...

Les objets Python sont détruits automatiquement par un *garbage collector* qui compte le nombre de références vers chaque objet.

Quand le compteur arrive à zéro, l'objet est détruit :

```
>>> class foo(object):
...     def __del__(self):
...         print("bye bye")
...
>>> e = foo()    # Py_INCREF()    --> 1
>>> f = []
>>> f.append(e)   # Py_INCREF()    --> 2
>>> del e         # Py_DECREF()    --> 1
>>> f.clear()     # Py_DECREF()    --> 0
bye bye
```

- ▶ Les attributs et méthodes « classiques » sont accessibles par la notation pointée.
- ▶ En revanche, les attributs préfixés par `__` sont inaccessibles en dehors de l'instance.  *attribut privé*

Des noms de méthodes sont réservés :

- ▶ `__init__(self)`, appelé lors de la création de l'objet;
- ▶ `__del__(self)`, appelé lors de la destruction de l'objet;
- ▶ `__repr__(self)`, appelé lors de l'affichage dans le terminal;
- ▶ `__add__(self, other)`, appelé par l'opérateur +;
- ▶ `__lt__(self, other)`, appelé par l'opérateur <;
- ▶ `__next__(self)`, appelé lors d'un `for i in ...`;
- ▶ etc.

PROPERTIES, GETTERS & SETTERS

```
class Circle(object):
    def __init__(self, radius):
        self.r = radius

    @property
    def radius(self):
        return self.r

    @radius.setter
    def radius(self, value):
        self.r = value

    @property
    def area(self):
        return math.pi * self.r**2

    @area.setter
    def area(self, value):
        self.r = math.sqrt(value/math.pi)
```

```
>>> a = Circle(2)
>>> a.area
12.566370614359172
>>> a.radius = 3
>>> a.area
28.274333882308138
>>> a.area = 2
>>> a.radius
0.7978845608028654
```


INTERLUDE



- ▶ Trier par ordre croissant de surface des formes géométriques définies comme suit :
 - ▶ un cercle : son rayon ;
 - ▶ un triangle : ses trois sommets ;
 - ▶ un quadrilatère : ses quatre sommets (dans l'ordre)

```
"""
```

```
Make a list of geometrical shapes and order them by area.
```

```
>>> c = Circle(1)
```

```
>>> t = Triangle(0, 4, 3j)
```

```
>>> q = Quadrilateral(0, 2, 2+2.5j, 2.5j)
```

```
>>> sorted([c, t, q])
```

```
[Circle of area 3.14, Quadrilateral of area 5.00, Triangle of area 6.00]
```

```
"""
```

```
import math
```

```
class Shape(object):
```

```
    def __lt__(self, shape):
```

```
        return self.area() < shape.area()
```

```
    def __repr__(self):
```

```
        return "%s of area %.2f" % (type(self).__name__, self.area())
```

```
class Circle(Shape):
    """Define a circle by its radius.

    >>> c = Circle(2)
    >>> c.area()/math.pi
    4.0
    """
    def __init__(self, radius):
        assert radius > 0, "Only positive radius"
        self.r = radius
    def area(self):
        # not very spectacular
        return math.pi * self.r**2
```

```
class Triangle(Shape):
    """Define a triangle by three vertices.

    >>> t = Triangle(0, 4, 3j)
    >>> t.area()
    6.0
    """
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
        self.v1 = b - a
        self.v2 = c - a
    def area(self):
        # z1.conjugate() * z2 = dot(z1, z2) + cross(z1, z2) * j
        # area is half of the abs value of the cross product
        return abs((self.v1.conjugate() * self.v2).imag) / 2.
```

```
class Quadrilateral(Shape):
    """Define a quadrilateral by four vertices.

    >>> q = Quadrilateral(0, 2, 2+2.5j, 2.5j)
    >>> q.area()
    5.0
    """
    def __init__(self, a, b, c, d):
        # cut the quadrilateral into two triangles and sum the areas
        self.t1 = Triangle(a, b, c)
        self.t2 = Triangle(c, d, a)
    def area(self):
        return self.t1.area() + self.t2.area()
```

PYTHON AVANCÉ

Python permet d'appeler une fonction en passant ses arguments :

- ▶ sous la forme d'un dictionnaire :

☞ *keyword arguments*

```
>>> d = {'real': 3, 'imag': 5}
>>> complex(**d) # complex(real=3, imag=5)
(3+5j)
```

- ▶ sous la forme d'un tuple :

☞ *positional arguments*

```
>>> t = (3, 5)
>>> complex(*t) # complex(3, 5)
(3+5j)
```


La syntaxe « décorateur » utilise une notation à base de @ :

```
>>> def explicit_call(func):
...     def func_wrapper(*args, **kwargs):
...         print ("calling " + func.__name__)
...         return func(*args, **kwargs)
...     return func_wrapper
...
>>> @explicit_call
... def is_positive(number):
...     return number > 0
...
>>> is_positive(4)
calling is_positive
True
```

La notation est équivalente à :

```
>>> is_positive = explicit_call(is_positive)
```


"Premature optimization is the root of all evil in programming."

"If you optimize everything, you will always be unhappy."

"I've never been a good estimator of how long things are going to take."

— Donald Knuth

Pour analyser les performances d'un code :

- ▶ le module `time` ( `%timeit` avec IPython);
- ▶ le module `cProfile`, plus complet.

```
import cProfile
cProfile.run('main()', sort='time')
```

- ▶ Cython permet à la fois :
 - ▶ d'optimiser du code Python en le compilant pour une architecture spécifique de machine ;
 - ▶ de créer une interface Python pour appeler du code écrit dans un autre langage, via son API C/C++.
- ▶ Le notebook `05_cython.ipynb` contient une introduction interactive aux possibilités de Cython.

- ▶ Un mécanisme interne à Python (le GIL) ne permet pas aux *threads* qui manipulent des objets Python de s'exécuter en parallèle;
 - ▶ possible, mais sans gain de performance
- ▶ Les extensions C (notamment NumPy) peuvent désactiver le GIL pour paralléliser des traitements;
- ▶ Le *multiprocessing* est une alternative mais les espaces de variables de chaque processus sont isolés.
- ▶ Numpy, Cython, Numba sont des bibliothèques optimisées pour le traitement parallèle.

PyQt5 propose une interface vers l'API de Qt⁶ (Nokia).

- ▶ joli, convivial, portable,
- ▶ mais chronophage !

Le principe est de fournir des fonctions, ou *callbacks*, à appeler lors d'un événement (clic sur un bouton, etc.)

- ▶ Dropbox, QGIS, calibre (ebooks), etc.

⁶ prononcé *cute*

La livraison d'applications Python pose des problèmes d'intégration quant à :

- ▶ la version de Python choisie pour le développement;
- ▶ les bibliothèques utilisées (et leurs versions).

On peut préparer des applications qui *embarquent* un interpréteur Python avec les bibliothèques requises.

☞ *environnements virtuels*

- ▶ `pyinstaller` est une bonne option;
- ▶ voir `py2app` pour la production d'applications Mac OS X.

```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than **right** now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

ET MAINTENANT,
AU BOULOT !!