Programmer en Python

Python 3.x par la pratique

Renaud Blanch <blanch@imag.fr>

Université Joseph Fourier, Polytech'Grenoble & UFR IM^{2}AG

décembre 2014

Objectifs du cours

Après avoir suivi ce cours, vous saurez :

- **utiliser** le langage Python pour réaliser des scripts ;
- **programmer** en tirant partie des concepts avancés du langage (classes, exceptions, modules) ; et
- **explorer** la bibliothèque standard et les extensions existantes pour être efficace avec Python.

Organisation

24 heures de cours/travaux pratiques :

- 6 en semaine 51 (15 décembre) ;
- 6 en semaine 2 (5 janvier) ;
- 6 en semaine 4 (19 janvier) ; et
- 6 en semaine 6 (5 février).

Évaluation

Vous serez **évalués** de la manière suivante :

- contrôle continu
- examen final

Ressources en ligne

La **page** consacrée au cours est ici :
`<http://iihm.imag.fr/blanch/teaching/python3/>`.

Sur cette page sont **disponibles** :

- les **supports de cours** ;
- les sujets de **travaux pratiques** ; et
- le **code** donné en exemple.

Ressources en ligne (cont.)

Webographie

- *The Python Tutorial* <http://docs.python.org/py3k/tutorial/index.html>
- *Dive Into Python 3* <http://www.diveintopython3.net>
- *Online Python Tutor - Visualize program execution* <http://pythontutor.com/visualize.html>

Installation

Ce cours présente **Python 3.x**, la dernière version du langage.
Pour savoir si elle est intallée, dans un terminal faites :

```
% python3 --version
Python 3.3.2
```

Si cela ne marche pas, il vous faut l'installer :

- **linux**, utilisez le gestionnaire de paquets de votre distribution (*e.g.*, `% sudo apt-get install python3`) ;
- **Mac OS X, Windows**, utilisez les distributions binaires les plus récentes fournies sur le site de Python : <http://python.org/download/releases/>

Interprète Python

Il est temps de tester votre interprète en **mode interactif.** Une fois lancé, l'interprète **attend une ligne** de code en entrée, **l'évalue**, puis **affiche le résultat** de cette évaluation :

```
% python3
```

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 13:52:24)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information
>>> 1+1
2
>>> "Hello" + " " + "world!"
'Hello world!'
>>> ^D
```

Éditeur de texte

Configurez votre **éditeur de texte** favori pour qu'il :

- utilise le **codage UTF-8** pour les fichiers contenant du code Python (`*.py`) ; et
- insère **4 espaces** lorsque vous tapez une tabulation.

Si votre éditeur ne permet pas ces deux choses, il est encore temps d'en changer !

Script

Le code peut être placé dans un fichier texte qui devient un **script**.

```
1 #! /usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 print("Hello world!")
```

listing 1 : _00_hello.py

Script (cont.)

```
% python3 _00_hello.py
Hello world!

% chmod a+x _00_hello.py
% ./_00_hello.py
Hello world!
%
```

Python vu du ciel

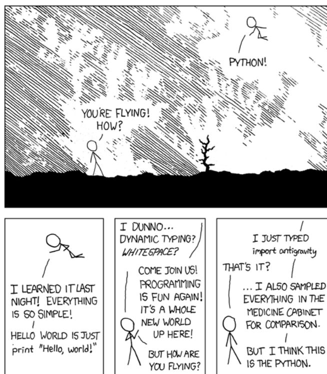

figure 1 : *"Perl, I'm living you"* `<http://xkcd.com/353/>`.

Caractéristiques

Python est un langage de programmation :

- **portable, simple et puissant** ;
- **interprété**, **typé dynamiquement** ;
- permettant divers **styles de programmation** (impératif, à objets, fonctionnel) ;
- offrant une **bibliothèque standard riche** ; et
- adapté à beaucoup de **classes de problèmes** grâce à des bibliothèques spécialisées proposées par une **communauté très active.**

Zen of Python

La "philosophie" du langage est résumée dans le "Zen of Python", accessible en tapant `import this` depuis l'invite interactive de Python.

```
>>> 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.
```

Zen of Python (cont.)

```
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!
```

Commentaires

Les **commentaires** commencent au caractère # et se prolongent jusqu'à la fin de la ligne.

```
>>> 7*6 # calcul du produit de 7 par 6
42
```

Indentation

La **structuration** du programme **en blocs** d'instructions est **donnée par l'indentation** et non par des balises explicites (*e.g.*, `{` et `}` pour les langages *à la* `C`).

```
>>> if 1 != 1:
...     print("We have a problem")
...
>>>
```

Variables

Les **variables** ne sont pas déclarées.
Elles sont (re)**définies par l'affectation** d'une valeur (avec =).

```
>>> a = 1+1 # l'affectation n'a pas de valeur, donc pas d'affichage
>>> a       # la variable a une valeur retournée par l'évaluation
2

>>> del a   # suppression d'une variable
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Typage

Les **variables** sont typées : elles **ont le type de leur valeur**. Celui-ci peut donc changer au cours du programme : c'est ce qu'on appelle le **typage dynamique**.

```
>>> a = "coucou"    # avec une chaîne de caractères dans a, ...
>>> type(a)         # ... a est de type "str" ...
<class 'str'>
>>> type("coucou")  # ... comme l'est son contenu.
<class 'str'>

>>> a = 42          # avec un entier dans a, ...
>>> type(a)         # ... son type devient "int".
<class 'int'>
```

Rien

Il existe en Python une valeur **nulle** (`None`), qui peut être utilisée pour dire qu'une variable n'a pas de valeur.

```
>>> type(None)
<class 'NoneType'>

>>> p = None
>>> p          # aucune valeur n'est retournée (et donc affichée)
```

Nombres

Les **entiers** sont de taille "infinie".

```
>>> type(42)
<class 'int'>
>>> 1234567**89
13957418598822621635241167796707938465481984077026315414890101571
14345875559162218718550539393707720856569152376896037670833499917
93713346471091572628494106076146419389211087459209170823939332677
8590937567344701723518950794307724881091056926484448002838012241
71409325694328243164300087409580250537167354213447294105206815597
15479472541631321425013768978692403188995380725532383634472359562
90854246954797114645706242235684570157179581172114612663973480990
45779670126089615289970121785372664201114467160014934769759348771
10973383995456863730247
```

Nombres (cont.)

Les nombres **décimaux** ont une précision limitée.

```
>>> type(3.14)
<class 'float'>
>>> 0.9999999999999999999
1.0
```

La division d'entiers donne un `float`
(même si elle tombe juste).

```
>>> 22/7
3.142857142857143
>>> 12/2
6.0
```

Booléens

Il y a deux **booléens** : `True` et `False`.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Les résultats des comparaisons sont des booléens.

```
>>> 1 == 2
False
>>> True != False
True
>>> 7 >= 5
True
```

Texte et données binaires

Le **texte** est stocké en unicode.

```
>>> type("Ça va ?")
<class 'str'>

>>> 'une chaîne de texte "avec des guillemets"'
'une chaîne de texte "avec des guillemets"'
>>> "avec l'apostrophe, les guillemets doivent être \"échappés\""
'avec l\'apostrophe, les guillemets doivent être "échappés"'
>>> """un retour
... à la ligne"""
'un retour\nà la ligne'
```

Texte et données binaires (cont.)

Les **chaînes d’octets** peuvent être données en ASCII.

```
>>> type(b"Content-Type: text/html; charset=utf-8")
<class 'bytes'>
```

N-uplet

Le **n-uplet** (`tuple`) permet de regrouper des valeurs pour en former une nouvelle.

```
>>> position = (45, 5)                    # groupage
>>> type(position)
<class 'tuple'>

>>> grenoble = ("Grenoble", position)  # groupage
>>> grenoble
('Grenoble', (45, 5))
>>> type(grenoble)
<class 'tuple'>
```

N-uplet (cont.)

```
>>> ville, (x, y) = grenoble            # dégroupage !
>>> x
45

>>> x, y = y, x                         # permutation !
>>> x, y
(5, 45)
```

Liste

La **liste** permet de stocker en séquence ordonnée plusieurs valeurs.

```
>>> names = ["pim", "pam", "pom"]
>>> type(names)
<class 'list'>

>>> names[2]                    # accès aux éléments par leur indice
'pom'
>>> names[2] = "poum"
>>> names
['pim', 'pam', 'poum']
```

Liste (cont.)

```
>>> names[-1]                    # names[-k] <=> names[len(names)-k]
'poum'
>>> names[1:-1]                  # borne inf. incluse, sup. exclue
['pam']

>>> names += ["toto"]            # concaténation de listes
>>> names
['pim', 'pam', 'poum', 'toto']

>>> del names[2]                 # suppression d'un élément
['pim', 'pam', 'toto']
```

Dictionnaire

Le **dictionnaire** est un tableau associatif, il permet de stocker des valeurs et de les retrouver à partir de clefs qui sont, elles aussi, des valeurs.

```
>>> months = { 1:"janvier",  2:"février",   3:"mars",
...            4:"avril",    5:"mai",       6:"juin",
...            7:"juillet",  8:"août",      9:"septembre",
...           10:"octobre", 11:"novembre", 12:"décembre"}
>>> type(months)
<class 'dict'>

>>> months[11]
'novembre'
```

Dictionnaire (cont.)

```
>>> z = {}           # un dictionnaire vide
>>> z[0, 0] = 4      # les clés peuvent aussi être des tuples ...
>>> z[1, 0] = 7      # ... ce qui permet de créer des tableaux ...
>>> z[1, 1] = 12     # ... à plusieurs dimensions.
>>> z
{(1, 0): 7, (0, 0): 4, (1, 1): 12}
>>> i, j = 1, 1
>>> z[i, j]
12
```

Ensemble

Un **ensemble** permet de stocker plusieurs valeurs **distinctes**.

```
>>> A = {1, 2, 3, 3, 2, 1}
>>> type(A)
<class 'set'>

>>> A
{1, 2, 3}
>>> B = {2, 4}

>>> A & B
{2}
>>> A | B
{1, 2, 3, 4}
>>> A ^ B
{1, 3, 4}
```

Condition

Les tests de **conditions** penvent s'enchaîner.

```
>>> t = 12
>>> if t < 10:
...     temperature = "froid"
... elif t < 20:
...     temperature = "tiède"
... else:
...     temperature = "chaud"
...
>>> temperature
'tiède'
```

Condition (cont.)

```
>>> if 2 in {1, 2, 3} & {2, 4}:
...     print("trouvé")
...
trouvé
```

Itération

On peut **itérer** sur un ensemble connu de valeurs.

```
>>> found = False
>>> for name in ["pim", "pam", "poum", "toto"]:
...     if name == "toto":
...         found = True
...
>>> found
True
```

Itération (cont.)

Il est possible de **passer** certaines itérations ou d'**interrompre** complètement le parcours des valeurs.

```
>>> for name in ["toto", "pim", "pam", "poum"]:
...     if len(name) != 3:
...         continue
...     if "m" in name:
...         break
...
>>> name
'pim'
```

Itération (cont.)

Beaucoup de types supportent l'itération : les dictionnaires, le texte, etc.

```
>>> for m in months:
...     if m > 6:
...         break
...     print(m, months[m])
...
1 janvier
2 février
3 mars
4 avril
5 mai
6 juin
```

Itération (cont.)

```
>>> for c in "ok":
...     print(c, ord(c))
...
o 111
k 107

>>> for b in b"ko":
...     print(b, chr(b))
...
107 k
111 o
```

Boucle

On peut également exécuter un bloc en **boucle** tant qu'une condition est vérifiée.

```
>>> t = 12
>>> while t < 20:
...     t += 1
...
>>> t
20
```

Définition

Python permet de définir des **fonctions** et des procédures. La seule différence entre les deux tient en ce qu'elles retournent ou non une valeur. Une procédure, sans `return` explicite donc, renvoie en fait `None` qui peut simplement être ignoré.

```
>>> def create_point(x, y, z):
...     return (x, y, z)
...
>>> create_point(4, 6, 3)
(4, 6, 3)
```

Valeurs par défaut

On peut donner des **valeurs par défaut** aux arguments d'une fonction.

```
>>> def create_point(x=0, y=0, z=0):
...     return (x, y, z)
...
>>> create_point(1, 2)
(1, 2, 0)
```

Arguments nommés

Le passage d'argument peut se faire en **nommant explicitement** les arguments auxquels on donne une valeur, ce qui permet de ne pas utiliser la valeur par défaut de l'un d'entre-eux tout en utilisant les autres.

```
>>> create_point(z=7, y=5)
(0, 5, 7)
```

Arguments groupés

On peut passer un **n-uplet** regroupant des **arguments** à une fonction **sans le dégrouper**.

```
>>> p = (4, 5)
>>> create_point(*p)
(4, 5, 0)
```

On peut également passer **un dictionnaire d'arguments nommés**.

```
>>> d = {"z": 3, "x":7}
>>> create_point(**d)
(7, 0, 3)
```

Listes d’arguments variable

Symétriquement, une fonction peut accepter un **nombre variable d’arguments**, nommés ou non.

```
>>> def varargs(*args, **kwargs):
...     return args, kwargs
>>> varargs(1, 2, c=3, a=4)
((1, 2), {'a': 4, 'c': 3})
```

Documentation

Les fonctions peuvent (doivent) être **documentées** en insérant des chaînes de caractères juste après leurs définitions.
Par **convention**, on indique brièvement sur la première ligne du commentaire l'utilité de la fonction, et on développe éventuellement sa description après avoir passé une ligne.
L'intérêt de cette documentation est qu'elle est accessible par la fonction `help`.

```
>>> def create_point(x=0, y=0, z=0):
...     """Create a 3D point."""
...     return (x, y, z)
...
```

Documentation (cont.)

```
>>> def create_point(x=0, y=0, z=0):
...     """Create a 3D point.
...
...     The function creates a point with its arguments:
...     x -- first coordinate
...     y -- second coordinate
...     z -- last coordinate
...     """
...     return (x, y, z)
...
>>> help(create_point)
```

Aide

L'**aide** s'affiche dans un nouvel écran, il faut taper sur `q` pour en sortir (`h` pour avoir la liste des commandes accessibles dans l'aide).

```
Help on function create_point in module __main__:

create_point(x=0, y=0, z=0)
    Create a 3D point.

    The function creates a point with its arguments:
    x -- first coordinate
    y -- second coordinate
    z -- last coordinate
(END)
```

Tests

On peut documenter le **comportement attendu** d'une fonction sous la forme du journal d'une session interactive.

```
3   def greet(name):
4       """Builds a greeting message.
5
6       argument:
7       name -- person to greet
8
9       >>> greet("brian")
10      'Hello brian!'
11      """
12      return "Hello " + name + "!"
```

listing 2 : _01_greeting.py

Tests (cont.)

Et Python permet de **tester** si le code se comporte comme prévu par sa **documentation** :

```
% python3 -m doctest -v _01_greeting.py
Trying:
greet("brian")
Expecting:
    'Hello brian!'
ok
1 items had no tests:
    _01_greeting
1 items passed all tests:
   1 tests in _01_greeting.greet
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Les exceptions

Lorsqu'une erreur se produit, Python génère une **exception** qui, par défaut, **interrompt l'évaluation** en cours.

```
>>> r = 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> r
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'r' is not defined
```

Information sur l’erreur

Le **type** de l’exception renseigne sur la **nature du problème**. La **trace** associé à l’exception donne des informations sur le **contexte** dans lequel l’erreur s’est produite.

```
>>> def ratio(x, y):
...     return x/y
...
>>> r = ratio(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in ratio
ZeroDivisionError: division by zero
```

Prévention des erreurs

Dans les langages **sans exceptions**, on traite les cas **particuliers d'abord** puis le cas **général ensuite**.

```
def lbyl_ratio(x, y):
        """look before you leap"""
    if y == 0:
        return x*float("inf")
    else:
        return x/y
```

Traitement des erreurs

Avec les exceptions, on préfèrera une approche **optimiste** : on **essaie le cas général**, et on traite ensuite éventuellement les **exceptions.**

```
def eafp_ratio(x, y):
    """easier to ask forgiveness than permission"""
    try:
        return x/y
    except ZeroDivisionError:
        return x*float("inf")
```

Traitement des erreurs (cont.)

```
>>> eafp_ratio(3, 0)
inf
>>> eafp_ratio("2", 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in eafp_ratio
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Ce style de programmation est résumé par l'acronyme `<EAFP>` : "*easier to ask forgiveness than permission.*"

Création d’exception

Les exceptions se créent avec `raise`.

```
54 def roman(number):
55     """Convert a number from decimal to roman notation."""
56     if not (0 < number < MAX_ROMAN):
57         raise ArithmeticError("value not in expected range")
58
59     result = ""
60     for digit, chars in zip(format(number, "04"),
61                             ROMAN_CHARS):
62         for i in ROMAN_INDICIES[digit]:
63             result += chars[i]
64     return result
```

listing 3 : _02_roman.py

Création d’exception (cont.)

```
>>> roman(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "_02_roman.py", line 57, in roman
    raise ArithmeticError("value not in expected range")
ArithmeticError: value not in expected range
```

Modules

Vous pouvez **séparer** le code d'un programme **en plusieurs fichiers**. Cette séparation n'est pas forcée par la langage mais est une facilité offerte qu'il ne faut pas négliger. On peut choisir de regrouper dans un fichier tout ce qui traite d'un même sous-problème.

Modules (cont.)

```
 3  """Convert euros and francs."""
 4
 5  EURO = 6.55957
 6
 7  def francs2euros(f):
 8      """Converts francs to euros."""
 9      return f/EURO
10
11  def euros2francs(e):
12      """Converts euros to francs."""
13      return e*EURO
```

listing 4 : _03_euro.py

Importation

Le module peut être **chargé** à l'aide de la commande `import`.
Ses fonctions sont alors accessibles à l'intérieur de l'**espace de nom** (*namespace*) créé pour le module.

```
>>> # exécute le fichier _03_euro.py dans l'espace de nom euro
... import _03_euro as euro
>>> euro.euros2francs(12) # appel d'une fonction du module
78.71484
```

Contenu

On peut explorer le **contenu** d'un module à l'aide de `dir`.
On peut également consulter son **aide** avec `help`.

```
>>> dir(euro)
['EURO', '__builtins__', '__cached__', '__doc__', '__file__',
'__name__', '__package__', 'euros2francs', 'francs2euros']

>>> help(euro.francs2euros)

Help on function francs2euros in module _03_euro:

francs2euros(f)
    Converts francs to euros.
(END)
```

Exécution

Si on veut que du **code** soit **exécuté** quand le script est **interprété directement par Python**, mais qu'il soit **ignoré** quand il est **importé comme un module**, on peut utiliser le patron suivant :

```
4  def main(argv=[__name__]):
5      """executed when run as script, skipped on import"""
6      return 0
7
8  if __name__ == "__main__":
9      import sys
10     sys.exit(main(sys.argv))
```

listing 5 : _04_main.py

Paquets

Les **modules** peuvent être placés dans des répertoires qui forment alors des **paquets**, à condition que ces répertoires contiennent un fichier `__init__.py` (qui sera exécuté lors du chargement de ce paquet).

Paquets (cont.)

```
% mkdir -p        package
% touch           package/__init__.py
% echo "R = 42" > package/module.py
% python3 -q
```

```
>>> import package.module
>>> package.module.R
42

>>> from package import module
>>> module.R = 12
>>> package.module.R
12
```

Le module builtins

Au **lancement** de Python, un certain nombre d'objets sont créés. Ceux-ci sont placés dans l'espace de nom `__builtins__` et sont accessibles directement. Ils offrent les fonctionnalités de base du langage.

Entrées/sorties

Les **entrées/sorties** peuvent se faire sur les **flux standards** avec `input` et `print`.
Pour des lectures/écritures simples dans des **fichiers**, on utilisera `open`.

```
>>> print("hello", input("name?\n"))
name?
bob
hello bob
>>> print(*open("test.txt"), sep="")
line 1
line 2
line 3
line 4
```

Calcul

La bibliothèque standard fournit quelques **fonctions mathématiques** :

- `all` et `any` calculent la **conjonction** (et) et la **disjonction** (ou) logique de leurs arguments ;
- `min`, `max` et `sum` calculent le minimum, le maximum et la somme d'un ensemble de valeurs ; et
- `round`, `abs`, `pow`, et `divmod` calculent l'**arrondi**, la **valeur absolue**, l'**élévation à une puissance** et le **quotient et le reste** de la division entière.

Calcul (cont.)

```
>>> all(x % 2 == 0 for x in [1, 2, 3, 4])
False
>>> any(x % 2 == 0 for x in [1, 2, 3, 4])
True

>>> sum([1, 2, 3, 4]) - 4*(4+1)/2
0.0
>>> round(123.45)
123

>>> q, r = divmod(42, 12)
>>> q, r, q*12 + r
(3, 6, 42)
```

Représentation textuelle

`repr` retourne une **représentation textuelle** de n'importe qu'elle valeur sous forme d'expression et `format` permet de **mettre en forme** des valeurs, en particulier numériques, mais pas uniquement.

```
>>> format(22/7, "8.3f")
'   3.143'
>>> format("Hello", "~<20")
'Hello~~~~~~~~~~~~~~~'
>>> format("Hello", "-^20")
'-------Hello--------'
>>> format("Hello", "=>20")
'===============Hello'
```

Représentation textuelle (cont.)

On peut obtenir une représentation **binaire**, **octale** ou **hexadécimale** des entiers avec `bin`, `oct` et `hex`.
On peut obtenir un **caractère** à partir de son **code** et réciproquement avec `chr` et `ord`.

```
>>> bin(42), oct(42), hex(42)
('0b101010', '0o52', '0x2a')

>>> list(b"toto") == [ord(c) for c in "toto"]
True
>>> chr(10)
'\n'
```

Manipulation de séquences

On peut connaître la **longueur** d'une séquence avec `len`.
On peut créer un **itérateur** à partir d'une séquence avec `iter`, et itérer sur ses valeurs avec `next`.

```
>>> vowels = "aeiou"
>>> len(vowels)
5

>>> i = iter(vowels)
>>> next(i), next(i), next(i)
('a', 'e', 'i')
>>> next(i), next(i), next(i, None)
('o', 'u', None)
```

Manipulation de séquences (cont.)

On peut :

- créer une séquence d'**entiers contigus** avec `range` ;
- **retourner** ou **trier** une séquence avec `reversed` et `sorted` ;
- **itérer sur une séquence** en donnant **également l'indice** des éléments avec `enumerate` ;
- **itérer sur plusieurs séquences** à la fois avec `zip` ; et
- **filtrer** ou **appliquer une fonction** à chaque élément d'une séquence avec `filter` et `map`.

Manipulation de séquences (cont.)

```
>>> l = list(range(4))
>>> r = list(reversed(l))
>>> l, r
([0, 1, 2, 3], [3, 2, 1, 0])
>>> sorted(l) == sorted(r)
True

>>> list(zip(l, r))
[(0, 3), (1, 2), (2, 1), (3, 0)]
>>> list(enumerate(r))
[(0, 3), (1, 2), (2, 1), (3, 0)]
>>> list(enumerate(l))
[(0, 0), (1, 1), (2, 2), (3, 3)]
```

Manipulation de séquences (cont.)

```
>>> def even(i): return i % 2 == 0
...

>>> list(map(even, l))
[True, False, True, False]
>>> list(even(i) for i in l)
[True, False, True, False]

>>> list(filter(even, l))
[0, 2]
>>> list(i for i in l if even(i))
[0, 2]
```

Support au typage

On trouve dans les *builtins* les **types** primitifs : `int`, `float`, `bool`, `str`, `bytes`, `tuple`, `list`, `dict`, `set`, etc.
On y trouve également des fonctions qui permettent d'obtenir des **informations sur les types** :

- `type` donne le **type** des valeurs ;
- `isinstance` (resp. `issubclass`) permet de tester l'**appartenance à une classe** d'une instance (resp. classe).

Support au typage (cont.)

```
>>> type(2)
<class 'int'>
>>> isinstance(3, int), isinstance(3, float)
(True, False)
>>> issubclass(int, object)
True
```

Interaction avec l’interprète

On peut explorer interactivement l’état de l’interprète :

- `help` donne la **documentation** d’un objet ;
- `dir` liste les **attributs** d’un objet.

La variable nommée _ stocke le **résultat de la dernière évaluation** de l’interprète :

```
>>> 7 * 2
14
>>> _ * 3
42
```

Interaction avec le compilateur

On peut **évaluer** (`eval`) un expression, **compiler** et **exécuter** du code (`compile`, `exec`).
On peut enfin accéder aux **variables locales** et **globales** présentes (`locals`, `globals`).

```
>>> a = 2
>>> eval("a * 3")
6
>>> globals()
{'a': 2, '__builtins__': <module 'builtins' (built-in)>,
'__package__': None, '__cached__': None, '__name__': '__main__',
'__doc__': None}
```

math

Le module `math` fournit des constantes et des fonctions mathématiques classiques :

- `e` et `pi` la **base des logarithmes naturels** et le **nombre** π ;
- les fonctions **trigonométriques** usuelles `cos`, `sin`, `tan` et leurs variantes (`cosh`, `acos`, `acosh`, etc.) ;
- `degrees` et `radians` qui convertissent les angles de **radians à degrés** et réciproquement ;
- `hypot` et `atan2` qui permettent de passer de coordonnées **cartésiennes à polaires** ;

math (cont.)

- `floor`, `ceil`, `trunc` qui **arrondissent** vers le bas, le haut ou tronque les nombres ;
- `exp`, `pow`, `log`, `log10`, `log1p` qui font les transformations en **exponentielles** et **logarithmes** ;
- etc.

sys

Le module `sys` permet en particulier d'**interagir avec le système** hôte de l'interprète Python. Il fournit par exemple les valeurs suivantes :

- `argv` contient une liste des **paramètres de la ligne de commande** ;
- `stdin`, `stdout`, `stderr` les **flux** d'entrée/sorties standards ;
- `maxsize` le **plus grand entier** représentable par l'architecture sous-jascente ;
- `platform` le **nom de la plateforme** sous-jascente (plus de détail dans le module `platform`) ; et
- `exit` est une fonction qui interrompt l'interprète et passe une valeur de retour au système.

sys (cont.)

```
4  def main(argv=[__name__]):
5      """executed when run as script, skipped on import"""
6      return 0
7
8  if __name__ == "__main__":
9      import sys
10     sys.exit(main(sys.argv))
```

listing 6 : _04_main.py

Classes

Le mot clef `class` permet de définir une **classe**. On peut mettre dedans des **méthodes** qui prennent explicitement en premier paramètre l'instance, notée `self`. La méthode `__init__` sert de **constructeur** aux instances.

```
>>> class Rectangle:
...     def __init__(self, width, height):
...         # les attributs de l'objet sont initialisés
...         # avec les valeurs des arguments du constructeur
...         self.width = width
...         self.height = height
...     def area(self):                     # définition d'une méthode.
...         """Compute rectangle area."""       # calcul à l'aide des
...         return self.width * self.height # attributs de l'instance
```

Instances

On peut alors créer des instances de la classe, utiliser ses méthodes et accéder à ses attributs :

```
>>> r = Rectangle(10, 7)              # création d’une instance
>>> r.area()                    # self est passé automatiquement
70
>>> r.width = 23             # les attributs sont toujours publics
>>> r.area()
161
```

Héritage

Comme dans tout langage objet, on peut **spécialiser** des classes pour enrichir leur comportement. Le typage étant dynamique, toutes les méthodes sont "virtuelles".

```
>>> class Shape:
...     """An abstract shape."""
...     def __init__(self, position):
...         self.position = position
...     def area(self):              # méthode virtuelle pure réalisée
...         raise NotImplementedError       # en levant une exception
...
>>> s = Shape((12, 34))    # la classe n'est pas vraiment abstraite,
>>> s.position                  # on peut l'instancier et l'utiliser
(12, 34)
```

Héritage (cont.)

```
>>> s.area()                    # sauf la partie non-implémentée
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "shape.py", line 7, in area
    raise NotImplementedError
NotImplementedError
```

Héritage (cont.)

```
>>> class Rectangle(Shape):                              # spécialisation
...     """A concrete rectangle inheriting from Shape."""
...     def __init__(self, position, width, height):
...         super().__init__(position)    # "super"-constructeur
...         self.width = width
...         self.height = height
...     def area(self):                                  # réalisation
...         return self.width * self.height
...
>>> r = Rectangle((12, 34), 56, 78)
>>> r.area()
4368
>>> r.position
(12, 34)
```

Héritage multiple

L'**héritage multiple** est **possible** en précisant plusieurs classes de base dans la définition de la classe. Cependant, cette possibilité est très peu utilisée en Python car le typage dynamique permet d'exploiter le **polymorphisme sans** nécessairement utiliser l'**héritage**.

Duck-typing

L’idée du `<duck-typing>` est que *“If it looks like a duck and quacks like a duck, it must be a duck”*.
Le principe est qu’**implémenter les méthodes** d’une interface (au sens de Java) **suffit** pour pouvoir les appeler. Il n’y a **pas besoin de dériver d’une classe de base** qui matérialise cette interface.

Duck-typing (cont.)

Concrètement, en programmation à la **C++** ou à la **Java**, on ferait :

```
>>> class Shape:     # classe "abstraite" spécifiant une interface
...     def area(self):
...         raise NotImplementedError
...
>>> class Rectangle(Shape):
...     def area(self):
...         return "rectangle area: width * height"
...
>>> class Circle(Shape):
...     def area(self):
...         return "circle area: radius * radius * pi"
```

Duck-typing (cont.)

Alors qu'en **Python** on peut simplement faire :

```
>>> class Rectangle:
...     def area(self): return "rectangle area"
...
>>> class Circle:
...     def area(self): return "circle area"
...
>>> for shape in [Rectangle(), Circle()]:
...     print(shape.area())                  # appel polymorphe
...
rectangle area
circle area
```

Durée de vie des objets

Comme en Java, et contrairement au C++, c'est **Python** qui **gère la mémoire** à l'aide d'un "**ramasse-miette**" *(garbage collector)*. Les objets sont détruits quand plus aucune référence ne permet d'y accéder.

Aucune garantie n'est donnée quant au **temps** après lequel un objet inatteignable est détruit. Lors de la destruction de l'objet, une **méthode de finalisation** est appelée pour permettre de libérer les ressources.

Cette méthode joue le même rôle qu'un destructeur mais le fait qu'on ne peut prévoir le moment où elle sera appelée interdit de s'en servir pour gérer des ressources automatiquement comme on le ferait en C++ (idiome du `<RAII>`).

Durée de vie des objets (cont.)

```
>>> class Test:
...     def __del__(self):
...         """Test finalizer."""
...         print("I'm dying!")
...
>>> t1 = Test()  # une instance de Test est crée
>>> t2 = t1      # t2 permet aussi d'accéder à cette instance
>>> t1 = None    # t1 ne permet plus d'accéder à l'instance
>>> t2 = None    # plus aucune variable ne référence l'instance,
I'm dying!       # elle est susceptible d'être détruite ce qui
>>>              # arrive instantanément ici mais n'est pas garanti
```

Conventions de nommage

Les attributs ou méthodes dont **le nom commence par un _** sont considérées comme étant des **détails d'implémentation** des objets.
Rien n'empêche de les utiliser (il s'agit juste d'une convention), cependant si le développeur a pris la peine de les documenter ainsi, mieux vaut respecter son choix.

Méthodes spéciales

Les noms **commençant et finissant par __** sont **réservés** pour Python.
Ils sont utilisés en particulier pour les `<méthodes spéciales>`.