Lambda-Expressions

- Une *lambda-expression* est une fonction anonyme qui peut être utilisée partout où une fonction est attendue.
- En Python, une lambda-expression est introduite par le mot-clé *lambda*. Elle ne peut contenir qu'une seule expression.
- Une lambda-expression Python renvoie toujours la valeur de son expression (pas d'instruction *return*).
- Sa syntaxe est de la forme *lambda params : expression*. (voir plus loin)

Collections

Les listes

- Une liste Python ressemble à un tableau des autres langages. C'est une *collection ordonnée* d'objets de tous types. On la note entre crochets en séparant ses éléments par des virgules. Elle est du type *list*.
- Les listes Python sont des structures de données *modifiables* (on peut modifier leur contenu après leur création).
- Les éléments d'une même liste peuvent être de types différents.
- Les éléments sont accessibles via leurs *indices* notés entre crochets. Le premier indice est 0.
- Les indices négatifs permettent de parcourir une liste de droite à gauche (l'indice -1 est l'indice du dernier élément, -2 celui de l'avant-dernier, etc.).

Exemples

```
liste = [2, 10, "bla", 3.14]
type(liste)                  # <class 'list'>
len(liste)                   # renvoie 4
liste[4]                     # IndexError: list index out of range !
liste[3]                     # renvoie 3.14   (comme liste[-1])
liste[1]                     # renvoie 10     (comme liste[-3])
```


Les tranches

- Une tranche de liste est une portion de liste délimitée par deux indices. Une liste étant modifiable, une tranche de liste est également modifiable.
- La tranche `une_liste[deb:fin]` désigne la tranche de *une_liste* comprise entre les indices *deb* compris et *fin non compris*.
- *deb* ou *fin* (ou les deux) peuvent être omis. En ce cas, les valeurs par défaut seront, respectivement *0* et *len(une_liste)*. Donc `une_liste[:]` est la tranche contenant tous les élements de la liste.
- On peut également indiquer un pas de progression (qui est de 1 par défaut) : `une_liste[deb:fin:pas]`.

Exemples

```
liste = ["un", "deux", "trois", "quatre"]
liste[1:-1]                  # renvoie ['deux', 'trois']
liste[:3]                    # renvoie ["un", 'deux', 'trois']
liste[2:]                    # renvoie ['trois', 'quatre']
liste[-2:-1]                 # renvoie ['trois']
liste[-1:2]                  # renvoie [] (parce que -1 est "après" 2)
liste[-1:2:-1]               # renvoie ['quatre']
l1 = liste                   # l1 désigne la même liste
l2 = liste[:]                # l2 contient les mêmes éléments que liste
```

Modification d'une liste

- Le contenu d'une liste peut être modifié directement au moyen des indices et des tranches (voir exemples).
- La méthode *l1.append(elt)* ajoute *elt* à la fin de *l1*.
- La méthode *l1.extend(l2)* ajoute les éléments de *l2* à la fin de *l1*.
- La méthode *l1.insert(indice, elt)* ajoute *elt* avant *indice*.
- La méthode *l1.remove(elt)* supprime la première occurrence de *elt* dans *l1*.
- La méthode *l1.pop([indice])* renvoie et supprime l'élément à l'indice indiqué (par défaut, indice = -1).
- La méthode *l1.clear()* supprime tous les éléments de la liste.
- L'instruction *del* permet de supprimer des éléments ou des tranches.
- La méthode *l1.reverse()* renverse *l1*.
- La méthode *l1.sort(key=None, reverse=False)* trie *l1*. Le paramètre *key* peut être une fonction (ou une lambda) renvoyant le critère de tri.

Modification d'une liste

Exemples

```
liste = list(range(3, 7))       # ou [*range(3, 7)] (Python 3.5)   [3, 4, 5, 6]
liste[1] = 'hello'              # [3, 'hello', 5, 6]
liste[1:3] = 'bla'              # [3, 'b', 'l', 'a', 6]
liste[1:3] = ['bla']            # [3, 'bla', 'a', 6]
liste[1:3] = 42                 # TypeError: can only assign an iterable
liste.append([3, 7])            # [3, 'bla', 'a', 6, [3, 7]]
liste.extend([30, 70])          # [3, 'bla', 'a', 6, [3, 7], 30, 70]
liste.append(42)                # [3, 'bla', 'a', 6, [3, 7], 30, 70, 42]
liste.insert(4, 'truc')         # [3, 'bla', 'a', 6, 'truc', [3, 7], 30, 70, 42]
liste.insert(0, 1000)           # [1000, 3, 'bla', 'a', 6, 'truc', [3, 7], 30, 70, 42]
liste.remove(3)                 # [1000, 'bla', 'a', 6, 'truc', [3, 7], 30, 70, 42]
liste.pop()                     # renvoie 42 et liste = [1000, 'bla', 'a', 6, 'truc', [3, 7], 30, 70]
liste.reverse()                 # [70, 30, [3, 7], 'truc', 6, 'a', 'bla', 1000]
liste.sort()                    # impossible : les éléments ne sont pas comparables entre eux
del liste[2:3]                  # [70, 30, 'truc', 6, 'a', 'bla', 1000]
liste[2:6] = []                 # [70, 30, 1000]
liste.sort()                    # [30, 70, 1000]
liste.sort(reverse=True)        # [1000, 70, 30]
liste.clear()                   # []
```

Opérations non destructrices sur les listes

- La fonction *sorted(liste, key=None, reverse=False)* renvoie une copie de *liste* triée (en fait, *sorted* fonctionne avec tous les objets Python itérables). Les éléments de *liste* doivent être comparables entre eux.
- Les opérateurs *elt in liste* et *elt not in liste* permettent de tester l'appartenance d'un élément à une liste.
- L'opérateur *l1 + l2* renvoie la concaténation de *l1* et *l2*.
- L'opérateur *liste * nbre* permet d'initialiser une liste et de lui fixer une taille initiale (ce qui évitera les réallocations futures).
- Les fonctions *min(liste)* et *max(liste)* renvoient respectivement le plus petit et le plus grand élément de la liste (ses éléments doivent être comparables entre eux).
- La méthode *liste.index(elt)* renvoie l'indice de la première occurrence de *elt* dans *liste* (ou une exception si l'élément ne s'y trouve pas).
- La méthode *liste.count(elt)* renvoie le nombre d'occurrences de *elt* dans *liste*.

Listes en intension

- Une liste en intension est décrite par les propriétés que doivent satisfaire ses éléments (les anglais les appellent « comprehension lists »).
- La syntaxe d'une liste en intension est de la forme (la partie *if* est facultative) :

```
liste = [expression for variable in iterable if condition]
```

- Exemples :

```
nbres = [1, 2, 3, 4]
carres  = [ x * x for x in nbres ]              # [1, 4, 9, 16]
carres2 = [ x * x for x in nbres if x > 2 ]     # [9, 16]
bissextiles = [annee for annee in range(1960, 2010)
               if (annee % 4 == 0 and annee % 100 != 0) or (annee % 400 == 0)]
codes = [s + z + c for s in "MF" for z in "SLMX" for c in "BGW"
         if not (s == "F" and z == "X")]
```

Tuples

- Un tuple est une sorte de liste *non modifiable* : on ne peut pas lui ajouter/ôter d'éléments après sa création et on ne peut pas non plus les modifier.
- Un tuple est noté entre parenthèses.
- L'accès (en lecture seule...) à ses éléments (ou à une tranche du tuple) utilise les crochets, comme les listes.
- La fonction *list(un_tuple)* renvoie une liste à partir d'un tuple, tandis que la fonction *tuple(une_liste)* renvoie un tuple à partir d'une liste.

Exemples

```
x = ('a', 'b', 'c')
type(x)                          # <class 'tuple'>
x[2]                             # 'c'
x[1:]                            # ('b', 'c')
len(x)                           # 3
min(x)                           # 'a'
5 in x                           # False
'b' in x                         # True
x[2] = 'd'                       # TypeError: 'tuple' object does not support item assignment
x + x                            # ('a', 'b', 'c', 'a', 'b', 'c')
x * 2                            # ('a', 'b', 'c', 'a', 'b', 'c')
x, y = 3, 4                      # identique à (x, y) = (3, 4)
(x + y)                          # 7... ce n'est PAS un tuple
(x + y,)                         # (7,) c'est un tuple à UN élément
```


Chaînes de caractères

- Les chaînes (le type *str*) peuvent être considérées comme des listes de caractères Unicode *non modifiables*.
- Comme pour les listes, on peut donc utiliser des indices et des tranches (mais uniquement pour lire, pas pour modifier).
- La fonction *len* permet de connaître la longueur d'une chaîne.
- Les méthodes qui semblent modifier une chaîne (*upper*, par exemple) ne modifient pas la chaîne mais renvoient une valeur modifiée de celle-ci.
- Le module *string* fournit des constantes utiles : *whitespace*, *digits*, *ascii_letters*, etc.

Chaînes de caractères

Exemples

```
str1, str2 = "bonjour", "salut"
x = str1 + str2                     # x = "bonjoursalut"
x = '*' * 10                        # x = "**********"
x.upper()                           # renvoie 'BONJOUR'
"BLA".lower()                       # renvoie 'bla'
"BLA bli".title()                   # renvoie 'Bla Bli'
str1.find('nj')                     # renvoie 2  (idem str1.index)
str1.find('ob')                     # renvoie -1
str1.rfind('o')                     # renvoie 4 (idem str1.rindex)
str1.index('ob')                    # ValueError: substring not found
str1.count('o')                     # 2
str1.startswith('bo')               # True
str1.replace('o', '*')              # renvoie 'b*nj*ur'
"  bla  ".strip()                   # renvoie 'bla'
"  bla  ".rstrip()                  # renvoie '  bla'
"  bla  ".lstrip()                  # renvoie 'bla  '
type(str1)                          # <class 'str'>
x = str1.encode("utf_8")            # convertit str1 en objet bytes (suite d'octets)
type(x)                             # <class 'bytes'>
import string
string.ascii_letters                # 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
str1.translate(str1.maketrans('bj', '*+'))  # renvoie '*on+our'
```


Formatage des chaînes

- Avant Python 3.6, les chaînes pouvaient être formatées avec la méthode *format* ou avec l'opérateur *%* (ce dernier ne devrait plus être utilisé dans les nouveaux programmes).

Exemples

```
"{} est la somme de {} et {}".format(10, 7, 3)      # '10 est la somme de 7 et 3'
"{0} est la somme de {2} et {1}".format(10, 7, 3)   # '10 est la somme de 3 et 7'
"Un tiers = {}".format(1/3)                         # 'Un tiers = 0.3333333333333333'
"Un tiers = {:1.2}".format(1/3)                     # 'Un tiers = 0.33'
"%d est la somme de %d et %d" % (10, 7, 3)          # '10 est la somme de 7 et 3'
"Un tiers = %f" % ((1/3))                           # 'Un tiers = 0.333333'
"Un tiers = %1.2f" % ((1/3))                        # 'Un tiers = 0.33'
```

- Python 3.6 a ajouté les *chaînes formatées* qui permettent d'*interpoler* des expressions :

```
f"La variable val contient la valeur {val}"
f"Il y {nb} élève{'s' if nb > 1 else ''}"
f"Avec 2 chiffres après la virgule : {val:.2}"       # 0.33 si val vaut 1/3
```

Dictionnaires

- Un dictionnaire est un *tableau associatif* : chaque élément de ce tableau est accessible via sa *clé*.
- Une clé de dictionnaire peut être de n'importe quel type *hachable* et *non modifiable*, ce qui est notamment le cas des nombres, des chaînes et des tuples de valeurs hachables.
- Chaque clé du dictionnaire est *unique*.
- Les valeurs stockées dans le dictionnaire peuvent être de n'importe quel type. Contrairement aux valeurs des listes (qui ont des indices numériques séquentiels), elles ne sont pas ordonnées.
- Un dictionnaire est du type *dict*. Le dictionnaire vide est noté *{}*.
- L'accès (en lecture ou en écriture) à une valeur du dictionnaire utilise la notation entre crochets, comme les listes (sauf que l'on utilise une clé et non un indice).
- Alors que l'écriture à un indice inexistant d'une liste provoque une erreur, l'accès en écriture à une clé inexistante d'un dictionnaire crée une nouvelle entrée dans celui-ci.

Exemples d'utilisation

Exemples

```
en_to_fr = {}                           # création d'un dico vide
en_to_fr['red'] = 'rouge'               # nouvelles associations clé -> valeur
en_to_fr|'blue'] = 'bleu'
en_to_fr['green'] = 'vert'

print("red is", en_to_fr['red'])        # Affiche 'red is rouge'

fr_to_en = { 'rouge': 'red', 'vert': 'green', 'bleu': 'blue'}
len(fr_to_en)                           # 3

list(en_to_fr)                          # ['blue', 'red', 'green']
list(en_to_fr.keys())                   # idem
list(en_to_fr.values())                 # ['bleu', 'rouge', 'vert']
list(en_to_fr.items())                  # [('blue', 'bleu'), ('red', 'rouge'), ('green', 'vert')]

for color in en_to_fr.keys():
  print(en_to_fr[color], end=', ')      # Affiche 'rouge, bleu, vert'

for color in en_to_fr:
  print(en_to_fr[color], end=', ')              # idem

for (color, couleur) in en_to_fr.items():
  print(f"{color} en français se dit {couleur}")
```

Remarques

- Les méthodes *keys*, *values* et *items* ne renvoient pas des listes, mais des *vues* dynamiques. Si l'on veut obtenir des listes, il faut donc les convertir avec *list*.
- Ces vues peuvent être parcourues comme n'importe quelle séquence, avec des *for*, des *in*, etc.
- Ces vues ne sont pas ordonnées (mais on peut les trier...).
- Si un accès en écriture à une clé inexistante crée une nouvelle entrée, un accès en lecture à une clé inexistante provoque l'exception *KeyError* (donc toujours utiliser *in* pour tester avant la présence d'une clé, ou préférer la méthode *get*).
- La fonction *del* permet de supprimer une entrée de dictionnaire.
- Comme pour les listes, on peut construire un *dictionnaire en intension*.

Exemples

Exemples

```
en_to_fr['purple']                      # KeyError : 'purple'

if 'purple' in en_to_fr:
  print(en_to_fr['purple'])             # N'affichera donc rien...

print(en_to_fr.get('purple', 'inconnu'))  # Affichera 'inconnu'

for color in sorted(en_to_fr):          # Tri sur les clés
   print(en_to_fr[color], end=', ')     # Affichera 'bleu, vert, rouge'

del(en_to_fr['blue'])                   # Supprime une entrée

liste = [1, 2, 3, 4]
dico_carres = { cle: cle**2 for cle in liste }           # {1: 1, 2: 4, 3: 9, 4: 16}
dico_cubes = { cle: cle**3 for cle in liste if cle > 2}   # {3: 27, 4: 64}
```

Cas d'utilisation typiques

- *Compter les mots d'une phrase :* chaque nouveau mot devient une clé (avec une valeur de 1). Cette valeur est incrémentée à chaque nouvelle occurrence.

```
import re                  # Pour les expressions régulières

phrase = 'To be or not to be, that is the question'
occurrences = {}
for mot in re.split('\W+', phrase):
  mot = mot.lower()
  occurrences[mot] = occurrences.get(mot, 0) + 1

# Affichage du résultat
for mot in sorted(occurrences):
  print(f"Le mot {mot} apparaît {occurrences[mot]} fois")
```


Cas d'utilisation typiques

- *Utilisation comme cache :* on met dans un dictionnaire des résultats déjà calculés afin de ne pas devoir les refaire ensuite.

```
def fibo_cache(n):
    cache = {0:0, 1:1}
    def fibo_aux(n):
        if n not in cache:
            cache[n] = fibo_aux(n-2) + fibo_aux(n-1)
        return cache[n]
    return fibo_aux(n)

def fibo(n):
    return n if n <= 1 else fibo(n - 2) + fibo(n - 1)
```

- Comparaison des temps d'exécution de *fibo(35)* et *fibo_cache(35)* :

```
% python3 -m perf timeit -s 'import fibo' 'fibo.fibo(35)'
.....................
Mean +- std dev: 4.92 sec +- 0.42 sec

% python3 -m perf timeit -s 'import fibo' 'fibo.fibo_cache(35)'
.....................
Mean +- std dev: 16.8 us +- 0.8 us
```

Ensembles

- Un ensemble est une collection de données *non ordonnées* et *non dupliquées*. Comme les clés d'un dictionnaire, les valeurs d'un ensemble doivent être *hachables* et *immutables* (cas des nombres, des chaînes et des tuples, mais pas des listes, des dictionnaires et des ensembles eux-mêmes).
- Outre l'ajout d'élément (et leur suppression), les opérations sur les ensembles sont les tests d'appartenance et d'inclusion, l'union, l'intersection et la différence.
- En Python, les ensembles sont implémentés par la classe *set*, l'ajout d'un élément par la méthode *add*, sa suppression par la méthode *remove*, le test d'appartenance par les opérateurs *in* ou *not in*. Les opérations d'union, d'intersection et de différence symétrique sont, respectivement, assurées par les opérateurs `|`, `&` et `^` (ou par les méthodes *union*, *intersection* et *symmetric_difference*). La différence ensembliste est implémentée par l'opérateur `-` ou la méthode *difference* (voir la doc pour les autres opérations...).
- Comme pour les autres collections, la fonction *len* renvoie le nombre d'éléments (sa « cardinalité ») et la boucle *for* permet de parcourir ses éléments.
- L'ensemble vide se note *set()*.
- Comme pour les listes et les dictionnaires, on peut créer des *ensembles en intension*.

Ensembles et test d'appartenance

Les ensembles étant implémentés par des hachages, ils sont particulièrement adaptés aux tests d'appartenance :

```
% python3 -m timeit -s 'li = list(range(100))' '"x" in li'
100000 loops, best of 3: 2.17 usec per loop
% python3 -m timeit -s 'li = set(range(100))' '"x" in li'
10000000 loops, best of 3: 0.0305 usec per loop
```

Même le cas le plus favorable des listes est à peine meilleur que les ensembles :

```
% python3 -m timeit -s 'li = list(range(100))' '0 in li'
10000000 loops, best of 3: 0.0243 usec per loop
% python3 -m timeit -s 'li = set(range(100))' '0 in li'
10000000 loops, best of 3: 0.0319 usec per loop
```

Exemple

```
s = set([1, 3, 5, 7])
t = set([1, 2, 3, 4, 6, 8])
s.union(t)                    # set([1, 2, 3, 4, 5, 6, 7, 8])
s | t                         # idem
s & t                         # set([1, 3])
s - t                         # set([5, 7])
s ^ t                         # set([2, 4, 5, 6, 7, 8])
s.issubset(set(range(1,10))) # True
s.add(3)                      # s non modifié
s.remove(2)                   # KeyError
if 2 in s: s.remove(2)        # s non modifié
s.discard(2)                  # pas d'erreur et s non modifié

# On exploite le fait que les ensembles soient implémentés à l'aide de dict :

u = {1, 3, 4, 12}             # set([4, 3, 12, 1])
u = { e for e in range(1,20) if e % 2 == 0 } # ensemble en intension

# set appliqué à un dictionnaire renvoie l'ensemble de ses clés :
moi = {'prénom': 'Eric', 'nom': 'Jacoboni', 'age': 20}
champs = set(moi)             # set(['age', 'nom', 'prénom'])
```


Modules

Introduction

- Les modules permettent de découper un projet en plusieurs fichiers. Ils permettent surtout de réutiliser du code.
- Un module est un fichier contenant des définitions de fonctions, de classes et autres objets et éventuellement du code exécutable directement. Le nom du module correspond à son nom de fichier (qui porte généralement l'extension *.py*).
- Un module peut être écrit en Python ou en C/C++. Quel que soit le langage utilisé pour le coder, son utilisation sera ensuite la même.
- Les modules permettent également d'éviter les conflits de noms car on peut toujours préfixer le nom d'un objet par le nom du module dans lequel il est défini.
- Un module définit un *espace de noms* (via un dictionnaire).
- Pour utiliser un module dans un autre fichier Python, il faut utiliser l'instruction *import*.
- Pour optimiser le chargement des modules, Python stocke leur version précompilée dans le répertoire *__pycache__* sous la forme *module.version.pyc* (où *version* est la version de Python qui a compilé ce module).

Exemple

- Soit le fichier *geometrie.py* suivant :

```
""" geometrie : un exemple de module écrit en Python """

pi = 3.14159

def surface_cercle(rayon):
  """ surface_cercle(rayon) : renvoie la surface du cercle de rayon indiqué."""
  global pi
  return pi * rayon**2
```

- Exemples d'utilisation :

```
pi                                  # NameError: name 'pi' is not defined
surface_cercle(2)                   # NameError: name 'surface_cercle' is not defined

import geometrie
pi                                  # NameError: name 'pi' is not defined
geometrie.pi                        # 3.14159
geometrie.surface_cercle(2)         # 12.56636

geometrie.__doc__                   # ' geometrie : un exemple de module écrit en Python '
geometrie.surface_cercle.__doc__    # ' surface_cercle(rayon) : renvoie la surface du cercle de rayo

from geometrie import *             # Pas conseillé...
surface_cercle(2)                   # 12.56636

from geometrie import pi as mon_pi, surface_cercle as surf_cercle
# ou : import geometrie.pi as mon_pi, geometrie.surface_cercle as surf_cercle

surf_cercle(2)
mon_pi
```

Recherche des modules

- Python recherche les modules dans les répertoires énumérés dans la variable *path* du module *sys* :

```
import sys
print(sys.path)       # Liste de chaînes contenant les répertoires des modules
```

- Ces répertoires sont recherchés dans l'ordre : dès que le module est trouvé, la recherche s'arrête. Si le module n'est pas trouvé, Python produit une exception *ImportError*.
- Lorsque l'on exécute un script Python, le premier chemin apparaissant dans la liste *sys.path* est toujours celui du répertoire où se trouve ce script.
- Dans une session interactive (et donc avec iPython), le premier chemin est la chaîne vide, qui représente le répertoire d'où a été lancé la session.

Stockage de ses propres modules

Il y a plusieurs choix pour stocker ses propres modules :

- Les mettre dans l'un des répertoires de *sys.path*. C'est la solution apparemment la plus simple, mais il ne faut *jamais* l'utiliser sous peine de risquer d'écraser des modules prédéfinis...
- Les mettre dans le même répertoire que le programme qui les utilise. Cette solution convient dans le cas où ces modules ne sont utilisés que par ce programme.
- Les mettre dans un (ou plusieurs) répertoire(s) particulier(s) et ajouter ce(s) répertoire(s) à *sys.path* : soit en modifiant directement *sys.path* dans le programme utilisateur avant d'importer le(s) modules(s), soit en initialisant la variable shell *PYTHONPATH*, soit en créant un *paquetage* (voir le tutoriel ou le manuel de référence pour la création de paquetages).

Noms exportés et noms privés

- L'instruction *from module import ** importe tous les noms du module qui ne sont pas explicitement cachés (pratique à éviter car on ne sait plus à quel module appartient un nom).
- Pour cacher un nom, il suffit de le préfixer par un blanc souligné. Il ne sera alors jamais importé implicitement par *from module import ** mais restera accessible par les autres méthodes (voir exemple).
- La fonction *dir(module)* renvoie la liste des noms définis dans le module indiqué. On remarquera que certains noms sont spéciaux (ceux encadrés par deux blancs soulignés, comme *__doc__*).
- L'entrée *__name__* contient le nom du module (qui vaut *'__main__'* pour les scripts et les sessions interactives.
- L'entrée *__builtins__* contient tous les noms prédéfinis (noms des exceptions prédéfinies, nom spéciaux prédéfinis, noms des fonctions prédéfinies).

Noms exportés et noms privés

Soit le module *mon_test.py* suivant :

```
"""Module de test des noms"""

def f(x): return x
def _g(x): return x

val1 = 42
_val2 = 64
```

Exemple d'utilisation :

```
from mon_test import *

f(3)         # Ok, renvoie 3
_g(3)        # NameError : '_g' is not defined

val1         # Ok, 42
_val2        # NameError : '_val2' is not defined

import mon_test

dir(mon_test)   # ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
                  '__package__', '__spec__', '_g', '_val2', 'f', 'val1']

mon_test.__name__  # 'mon_test'

mon_test._g(3)  # Ok, renvoie 3
mon_test._val2  # Ok, 64

from mon_test import _g
_g(3)           # Ok, renvoie 3
```

__name__ et '__main__'

- Lorsque l'on écrit un module, il peut être utile d'y ajouter un code permettant de le tester directement.
- Ce code doit s'exécuter si le module est lancé comme un script (avec *python3 monmodule.py*, par exemple). Sa variable *__name__* vaudra alors *'__main__'*.
- Il ne doit pas s'exécuter si le module est importé (avec *import* ou *from*). Sa variable *__name__* contiendra alors le nom du module.
- Il suffit donc de tester dans le module si *__name__* est égal à *'__main__'* :

```
"""Un module de test... """

... définition du module ...

if __name__ == '__main__':
  ... code de test du module qui ne s'exécutera que si ce fichier est lancé comme un script
```

- Ceci ne dispense pas de l'écriture de tests unitaires... (*Cf.* les exemples d'utilisation de *nose*).

Le module zipapp

- Python peut directement exécuter des archives ZIP à condition que cette archive contienne un fichier *__main__.py*.
- En pratique, ceci signifie que l'on peut donc livrer une application Python sous la forme d'un unique fichier qui aura l'extension *.pyz*.
- La distribution de Python 3.5 contient un module *zipapp* permettant de créer une telle archive.
- La démarche consiste à :
  1. Regrouper dans une arborescence tous les fichiers de son application.
  2. Renommer le fichier principal en *__main__.py*.
  3. Créer l'archive en faisant *python3 -m zipapp nomrep* (ou *nomrep* est la racine de l'arborescence de l'application), ce qui aura pour effet de créer une archive *nomrep.pyz* (avec les versions précédentes de Python, il fallait créer l'archive « à la main »).
  4. Pour exécuter l'application, il suffit ensuite de faire *python3 nomrep.pyz*
- Pour en savoir plus, lire le PEP 0441.