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.

Classes et POO

Définition d'une classe

- Une classe simple est introduite par le mot-clé *class*, suivi du nom de la classe :

```
class NomClasse:
  ... corps de la classe : définition de méthodes, affectation de variables.
```

- Attention, par défaut les classes Python sont plutôt permissives (tout est public, on peut rajouter des champs à un objet, etc.) :

```
class Cercle:
  pass

un_cercle = Cercle()   # Crée un objet Cercle
un_cercle.rayon = 5    # On peut lui rajouter un attribut à la volée !
del un_cercle.rayon    # Ou en supprimer...
```

- Généralement, on ajoute la méthode spéciale *__init__* pour initialiser une instance lors de sa création (celle-ci joue donc le rôle des « constructeurs » de C++ ou Java) :

```
class Cercle:
  def __init__(self, rayon):   # Toutes les méthodes doivent avoir self comme premier paramètre
    self.rayon = rayon         # self.rayon est une variable d'instance, rayon est le paramètre

un_cercle = Cercle(5)   # Appel implicite de Cercle.__init__(un_cercle, 5)
```

Définition

Quelques remarques importantes à savoir lorsque l'on a pratiqué d'autres langages de POO :

- Les méthodes doivent avoir un premier paramètre *self* qui désignera l'instance au moment de l'appel. Ce paramètre est passé implicitement à l'appel de la méthode.
- Les variables d'instance doivent obligatoirement être préfixées par *self* afin de les distinguer des variables locales et des paramètres. Python étant un langage dynamique, les variables d'instances sont créées lors de leur première affectation.
- Par défaut, les méthodes et les variables sont publiques et les classes sont « ouvertes » (on peut leur rajouter/ôter des variables et des méthodes depuis l'extérieur).
- Mais le programmeur peut créer des méthodes et des variables privées, empêcher la modification d'une classe, etc.

Variables d'instance et variables de classe

- Dans le corps de la classe, les variables d'instances sont préfixées par *self* et sont créées lors de leur première affectation (généralement, dans *__init__*).
- Toute variable initialisée en dehors de toute méthode est considérée comme une variable de classe : elle existe même si aucune instance n'a été créée et sa valeur est partagée par toutes les instances de la classe.
- Pour accéder à une variable de classe, il faut préfixer son nom du nom de la classe.
- Attention à certains problèmes (voir l'exemple).

```
class Cercle:
  pi = 3.14159                # Variable de classe
  def __init__(self, rayon):  # Toutes les méthodes doivent avoir self comme premier paramètre
    self.rayon = rayon        # self.rayon est une variable d'instance, rayon est le paramètre

print(Cercle.pi)              # Affiche 3.14159
c = Cercle(2)                 # Crée un cercle de rayon 2
print(c.rayon)                # Affiche 2
print(c.pi)                   # Affiche 3.14159
c.pi = 12                     # Ouch : on vient de créer une variable d'instance pi pour l'objet c
print(c.pi)                   # Affiche 12...
print(Cercle.pi)              # Affiche 3.14159
```

Variables d'instance et variables de classe

Si l'on veut vraiment empêcher la création dynamique d'attributs d'instance, on peut utiliser la méthode spéciale `__setattr__` :

```
class Cercle:
  pi = 3.14159                # Variable de classe
  def __init__(self, rayon):  # Toutes les méthodes doivent avoir self comme premier paramètre
    self.rayon = rayon        # self.rayon est une variable d'instance, rayon est le paramètre

  def __setattr__(self, nom, val):  # Appelée à chaque affectation d'un attribut
    if nom not in ['rayon']:
       raise AttributeError(f"{nom} n'est pas défini")
    else:
       self.__dict__[nom] = val

c = Cercle(2)
c.pi = 12                     # AttributeError: pi n'est pas défini
c.pouet = 24                  # AttributeError: pouet n'est pas défini
c.rayon = 42                  # Ok
```

Mais ce n'est que rarement nécessaire...

Méthodes d'instance et de classe

- Alors que les méthodes d'instance s'appliquent à une instance particulière de la classe, les méthodes de classe s'appliquent à *toutes* les instances d'une classe.
- Python permet de créer à la fois des méthodes *de classe* et des méthodes *statiques*. Leur principale différence est la syntaxe de leur définition.
- Une méthode de classe ou une méthode statique peut être appelé sur le nom de la classe où sur celui d'une instance.
- Les décorateurs *@classmethod* et *@staticmethod* permettent, respectivement, de définir une méthode de classe et une méthode statique.
- Évidemment, une méthode statique ou de classe qui manipulerait une variable d'instance n'aurait aucun sens (alors qu'une méthode d'instance peut manipuler une variable de classe).

Exemple de méthode statique

Fichier cercle.py

```
class Cercle:
  # Variables de classe
  tous = []
  pi   = 3.14159

  def __init__(self, rayon):
     self.rayon = rayon
     Cercle.tous.append(self)   # On ajoute le cercle à la liste de tous les cercles
                                # Ou : self.__class__.tous.append(self)
  def surface(self):
     return Classe.pi * self.rayon**2

  @staticmethod
  def surface_totale():
     total = 0
     for c in Cercle.tous:
        total += c.surface()
     return total
```

Exemple d'utilisation :

```
import cercle
c = cercle.Cercle(4)
c2 = cercle.Cercle(3)
print(cercle.Cercle.surface_totale())        # 78.53975
```


Exemple de méthode de classe

Fichier cercle.py

```
class Cercle:
  ... idem précédemment ...

  @classmethod
  def surface_totale(cls):
     total = 0
     for c in cls.tous:
        total += c.surface()
     return total
```

- L'avantage d'utiliser une méthode de classe est que cette méthode connaît la classe via le paramètre qui lui a été passé (*cls*, ici). On peut donc écrire simplement *cls.tous* dans le corps de la méthode.
- Une méthode statique n'a pas ce paramètre et doit donc soit coder « en dur » le nom de la classe (*Cercle.tous*, dans notre exemple), soit passer par la variable spéciale *__class__*.
- Une méthode de classe peut également être redéfinie dans une sous-classe, pas une méthode statique... Mon conseil : utilisez plutôt des méthodes de classe.

Variables et méthodes privées

- Par défaut, tous les membres d'une classe sont *publics* : les variables d'instance notamment, peuvent être modifiées depuis l'extérieur, ce qui nuit au principe d'encapsulation des données.
- En Python, il n'existe pas de mot-clé *private* comme en Java, C++ ou C# : pour créer des variables et des méthodes privées, il suffit de préfixer leur nom par deux blancs soulignés.
- En réalité, l'attribut (ou la méthode) ne sont pas « privés » : leur nom est simplement modifié par Python pour compliquer leur accès depuis l'extérieur de la classe.
- Attention : un nom de membre privé ne doit pas se terminer par deux blancs soulignés (ce format est réservé aux noms spéciaux de Python, comme *__init__* ou *__doc__*).
- L'avantage de cette convention est qu'elle facilite l'identification des membres privés.

Exemple

Fichier point.py

```
import math

class Point:
  "Classe définissant un point du plan"

  def __init__(self, x, y):
    "Crée le point d'abscisse x et d'ordonnée y"
    self.__x, self.__y = x, y                    # __x et __y sont "privées"

  def getX(self): return self.__x
  def getY(self): return self.__y

  def __distance_origine(self):                  # __distance_origine est "privée"
     return math.hypot(self.__x, self.__y)

  def getDistance(self):
     "Renvoie la distance du point par rapport à l'origine (0,0)"
     return self.__distance_origine()
```

Exemple d'utilisation :

```
from point import Point
p = Point(3, 2)
p.__x                   # AttributeError: 'Point' object has no attribute '__x'
p.getX()                # 3
p.__distance_origine()  # AttributeError: 'Point' object has no attribute '__distance_origine'
p.getDistance()         # 3.605551275463989
```

Propriétés

- Au lieu d'utiliser des accesseurs Java-esque comme *getX()* ou *setX(valeur)* pour lire ou modifier des variables d'instances privées, il est préférable d'utiliser des *propriétés*.
- L'avantage des propriétés est qu'elles allègent le code utilisateur et qu'elles assurent le principe d'accès uniforme : l'utilisateur d'une classe n'a pas besoin de savoir si une information lui est fournie par une méthode ou par une variable.
- Ce mécanisme est également utilisé par d'autres langages, comme C# ou Ruby.
- Une propriété de lecture est introduite par le décorateur *@property*.
- Si l'on veut lui ajouter une propriété d'écriture, on ajoute à cette propriété un décorateur « setter » (voir exemple).

Exemple

```
import math

class Point:
  """Classe définissant un point du plan"""

  def __init__(self, x, y):
    """Crée le point d'abscisse x et d'ordonnée y"""
    self.__x, self.__y = x, y

  @property
  def x(self): return self.__x
  @x.setter
  def x(self, new_x): self.__x = new_x

  @property
  def y(self): return self.__y
  @y.setter
  def y(self, new_y): self.__y = new_y

  @property
  def distance(self): return math.hypot(self.__x, self.__y)
```

Exemple d'utilisation :

```
from point import Point
p = Point(3, 2)
p.x                     # 3 (utilisation de la propriété en lecture x)
p.y = 4                 # utilisation de la propriété en écriture y
p.distance              # 5.0 (utilisation de la propriété en lecture distance)
```

Héritage

- Une classe Python peut hériter d'une ou plusieurs classes. En réalité, les classes précédentes héritaient de *object*, la classe racine de la hiérarchie des classes Python.
- Contrairement à Java, Ruby ou C#, Python autorise l'héritage multiple : une classe peut hériter de plusieurs classes.
- Une classe hérite de tous les membres non privés de ses super-classes et elle peut redéfinir certaines méthodes.
- Un appel à *super().une_methode()* permet d'appeler la méthode *une_methode()* définie dans une super-classe (mais son utilisation est critiquée par certains, **notamment par moi**...).
- L'ordre de recherche des attributs part de la classe, puis remonte dans la première super-classe de la liste, puis remonte dans les super-classes de celle-ci. La recherche se poursuit avec la seconde super-classe, etc. On a donc une recherche de gauche à droite, en profondeur d'abord.
- La fonction *isinstance(obj, cls)* teste si un *obj* est une instance de *cls* ou de l'une de ses sous-classes.
- La fonction *issubclass(cls1, cls2)* teste si la classe *cls1* est une sous-classe de *cls2*.

Exemple

```
class PointNomme(Point):
  """Classe définissant un point étiqueté"""

  def __init__(self, x, y, nom):
     Point.__init__(self, x, y)            # Appel du constructeur de Point
     # Ou : super().__init__(x, y)
     self.__nom = nom

  @property
  def nom(self): return self.__nom
```

Exemple d'utilisation :

```
import point2

p = point2.PointNomme(3, 2, "Point 1")
p.nom                                  # 'Point 1'
p.x                                    # 3
p.distance                             # 3.605551275463989
p.y = 3
p.distance                             # 4.242640687119285
p.__class__                            # <class 'point2.PointNomme'>
```

Polymorphisme

Toutes les méthodes d'une classe Python sont virtuelles, ce qui signifie qu'elles peuvent être redéfinies dans les classes filles :

```
class ClasseBase:

  def __init__(self, x, y):
    self.x, self.y = x, y

  def deplace(self, delta_x, delta_y):
     self.efface()
     self.x += delta_x
     self.y += delta_y
     self.affiche()

  def efface(self):   # on ne sait pas encore le faire...
    pass              # Ou  ...  (en Python 3)

  def affiche(self):  # on ne sait pas encore le faire...
    pass

class Fille(ClasseBase):

   def __init__(self, x, y, nom):
      ClasseBase.__init__(self, x, y)      # Ou super().__init__(x, y)
      self.nom = nom

   def efface(self):
      print(f"{self.nom} s'efface...")

   def affiche(self):
      print(f"{self.nom} s'affiche...")
```


Polymorphisme

Exemple d'utilisation :

```
import polymorph

f = polymorph.Fille(10, 12, "fille")
f.x                                    # 10
f.y                                    # 12
f.deplace(10, 10)                      # deplace() est définie dans ClasseBase
                                       # Affiche 'fille s'efface...'
                                       # Affiche 'fille s'affiche...'
f.x                                    # 20
f.y                                    # 22
```

- L'appel *f.deplace(...)* a appelé la méthode *deplace()* de la super-classe.
- Comme la classe *Fille* redéfinit les méthodes *efface()* et *affiche()*, la méthode *deplace()* les appelle car ce sont elles qui sont « les plus proches » de l'objet *f*.
- On a donc un « double dispatch » : comme *Fille* ne définit pas *deplace()*, on remonte vers sa super-classe pour trouver *deplace()*, puis on redescend vers la classe de *f* pour trouver les méthodes *efface()* et *affiche()* les plus spécialisées.
- Techniquement, on n'a pas besoin de définir les méthodes *efface()* et *affiche()* dans *ClasseBase*. Mais, si on ne le fait pas, un appel à *deplace()* sur un objet de *ClasseBase* provoquera un *AttributeError*. Pour simuler une *classe abstraite*, on peut également implémenter ces deux méthodes pour qu'elles renvoient *NotImplemented*.

Représentation textuelle

- Les deux méthodes spéciales *__repr__* et *__str__* permettent de gérer la représentation textuelle d'un objet. Ces deux méthodes sont appelées automatiquement par les fonctions *repr()* et *str()*.
- Par défaut, toutes les deux produisent une chaîne décrivant l'objet : *<point3.Point object at 0x7f5a2f857910>*, par exemple.
- La méthode *__repr__* est censée produire une représentation textuelle de l'objet. Cette représentation n'est pas destinée à être lue par un humain, mais doit permettre de recréer l'objet via un appel à *eval()* (cf. exemple). Elle est appelée automatiquement par la fonction *repr(obj)* (et par idle ou l'interpréteur intéractif).
- La méthode *__str__* est censée produire une représentation lisible de l'objet. Elle est appelée automatiquement par les instructions comme *print* et par la fonction de conversion *str(obj)*.

Exemple

```
# Module point4.py

class Point:
  (...)
  def __repr__(self):
    return f"Point({self.x}, {self.y})"

  def __str__(self):
    return f"({self.x}, {self.y})"

class PointNomme(Point):
  (...)
  def __repr__(self):
    return f"PointNomme({self.x}, {self.y}, {self.nom})"

  def __str__(self):
    return f"({self.x}, {self.y}, {self.nom})"
```

Exemple d'utilisation :

```
from point4 import Point

p = Point(3, 2)
repr(p)                                 # Point(3, 2)
q = eval(repr(p))                       # exécute donc q = Point(3, 2)...
r = eval(p.__module__ + '.' + repr(p))  # Si on avait simplement fait 'import point4'
q == p                                  # True
print(p)                                # Affiche (3, 2)  (appel de str(p))
```

Comparaisons

- Par défaut, deux objets *Point* distincts seront toujours considérés comme différents (l'égalité par défaut compare les références...).
- En réalité, les opérateurs de comparaison (et les autres aussi, d'ailleurs) appellent des méthodes spéciales d'*object* que l'on peut redéfinir pour modifier leurs comportements.
- Dans le cas des opérateurs de comparaison, il s'agit des méthodes *__eq__*, *__ne__*, *__lt__*, *__le__*, *__gt__* et *__ge__*.
- Si l'on redéfinit *__eq__*, Python redéfinit automatiquement *__ne__* si on ne le fait pas (mais autant le faire explicitement).
- Pour les comparaisons, une bonne pratique consiste à redéfinir tous les opérateurs afin qu'il n'y ait pas d'incohérences. Afin de faciliter, cette tâche, on dispose de *functools.total_ordering* qui les génère tous à partir de *__eq__* et de l'un des opérateurs *__lt__*, *__le__*, *__gt__* et *__ge__*.
- L'instruction *obj1 == obj2* appelle en réalité *obj1.__eq__(obj2)* (idem pour les autres fonctions de comparaison).

Exemple

```
import math
from functools import total_ordering

@total_ordering
class Point:
  def __init__(self, x, y):
    self.__x, self.__y = x, y

  def __eq__(self, autre):
    if not isinstance(autre, Point):
      return NotImplemented          # Python essaiera de trouver autre.__eq__(self)
    else:
      return self.__x == autre.x and self.__y == autre.y

  def __lt__(self, autre):
    if not isinstance(autre, Point):
      return NotImplemented          # Python essaiera de trouver autre.__lt__(self)
    else:
      return self.distance < autre.distance

  @property
  def x(self): return self.__x
  @x.setter
  def x(self, new_x): self.__x = new_x

  @property
  def y(self): return self.__y
  @y.setter
  def y(self, new_y): self.__y = new_y

  @property
  def distance(self): return math.hypot(self.__x, self.__y)
```

Exemple (suite)

```
@total_ordering
class PointNomme(Point):
  """Classe définissant un point étiqueté"""

  def __init__(self, x, y, nom):
     Point.__init__(self, x, y)         # Ou super().__init__(x, y)
     self.__nom = nom

  def __eq__(self, autre):
    return Point.__eq__(self, autre) and self.__nom == autre.nom
    # ou return super().__eq__(autre) and self.__nom == autre.nom

  def __lt__(self, autre):
    return Point.__lt__(self, autre) and self.__nom < autre.nom
    # ou return super().__lt__(autre) or self.__nom < autre.nom

  @property
  def nom(self): return self.__nom
```

Tests :

```
p = point3.Point(3, 2)
q = point3.Point(3, 2)
r = point3.Point(4, 6)
p == q                          # True
p != r                          # True
p < r                           # True
p > r                           # False
p <= q                          # True
```


La méthode spéciale __hash__

- On rappelle que pour pouvoir servir de clé, un objet doit être immutable et disposer d'une méthode *__hash__* (qui est appelée automatiquement par la fonction *hash(obj)*).
- Les objets *Point* ne sont pas immutables (pour qu'ils le soient, il suffit de supprimer les deux propriétés setters qui permettent de modifier les attributs du point). Voir l'exemple suivant.
- Si l'on redéfinit la méthode spéciale *__eq__*, Python ne fournit plus de méthode *__hash__* par défaut : les objets de notre classe ne peuvent donc plus servir de clé de dictionnaire.
- La création d'une méthode de hachage est un problème épineux car il faut s'assurer qu'elle fournira une valeur différente pour chaque objet différent au sens de *__eq__*.
- Dans le cas d'un point, notamment, il faut que le hachage d'un *Point(3, 2)* soit différent du hachage d'un *Point(2, 3)* et que *tous* les *Point(3, 2)* renverront la même valeur de hachage.
- Pour écrire une nouvelle fonction de hachage, le plus simple (et le plus sûr) consiste à utiliser une fonction de hachage existante.

Exemple

```
# Module point5.py

import math
from functools import total_ordering

@total_ordering
class Point:
  """Classe définissant un point du plan"""
  (...)
  @property
  def x(self): return self.__x      # On a supprimé le setter

  @property
  def y(self): return self.__y      # On a supprimé le setter

  @property
  def distance(self): return math.hypot(self.__x, self.__y)

  def __hash__(self):
   return hash((self.__x, self.__y))  # le hachage du tuple (x,y) sera différent de celui de (y, x)

@total_ordering
class PointNomme(Point):
  """Classe définissant un point étiqueté"""
  (...)
  @property
  def nom(self): return self.__nom

  def __hash__(self):
    return hash((self.__x, self.__y, self.__nom))
```


Exemple

La fonction prédéfinie *hash(obj)* appelle la méthode *obj.__hash__()* :

```
from point5 import Point

p = Point(3, 2)
q = Point(3, 2)
r = Point(2, 3)

id(p)                       # 139751287446864
id(q)                       # 139751287446672 (donc différent alors que p == q)
hash(p)                     # 3713083796995235906
hash(q)                     # 3713083796995235906 (donc égal)
hash(r)                     # 3713082714463740756

d = {p: "toto", r: "titi"}  # {Point(3, 2): 'toto', Point(2, 3): 'titi'}
d[q] = "normal"
d                           # {Point(3, 2): 'normal', Point(2, 3): 'titi'}
p = Point(1, 1)
d                           # {Point(3, 2): 'normal...', Point(2, 3): 'titi'}
d[p] = "autre"
d                           # {Point(1, 2): 'autre', Point(3, 2): 'normal...', Point(2, 3): 'titi'}
```

Remarque : la fonction *id(obj)* renvoie l'adresse mémoire de l'objet concerné. C'est donc un moyen d'identifier de façon unique deux objets différents mais elle considérera aussi comme différents deux points qui ont pourtant les mêmes coordonnées et qui, du point de vue d'un hachage, devraient être considérés comme égaux.

Création d'une classe collection

- Le but, ici, est de créer une classe « collection » qui se comporte comme les classes collections prédéfinies (*list*, *dict*, *tuple*)...
- On veut notamment pouvoir accéder en lecture (et en écriture, si notre classe n'est pas immutable) à un élément en utilisant la notation des crochets et on veut pouvoir parcourir tous les éléments de notre collection à l'aide d'une boucle *for*.
- Les méthodes spéciales *__getitem__*, *__setitem__* et *__delitem__* permettent de bénéficier de la notation entre crochets.
- La méthode *__contains__* permettent d'utiliser les opérateurs *in* et *not in* pour tester la présence d'un élément dans une collection.
- Les méthodes spéciales *__add__*, *__mul__* et *__len__* permettent d'implémenter, respectivement, la concaténation, la multiplication et la longueur d'une collection.
- La méthode spéciale *__iter__* permet de parcourir les éléments d'une collection. Elle est appelée automatiquement par *for*.

Premier exemple : un type ensemble

```
class Ensemble:
  """ Implémentation d'un ensemble d'entiers à l'aide d'un hachage
      dont les clés sont les entiers et les valeurs des booléens """

  def __init__(self, *liste):
     self.__corps = {}
     for e in liste: self.__corps[e] = True

  def contient(self, elt):
    return self.__corps.get(elt, False)   # ou : return elt in self.__corps.keys()

  def ajoute(self, elt):
    self.__corps[elt] = True

  def supprime(self, elt):
    if self.__corps.get(elt): del self.__corps[elt]

  def __str__(self):
    s = ""
    for e in self.__corps.keys(): s += str(e) + ", "
    return s[:-2]

  def union(self, autre):
    if not isinstance(autre, Ensemble): return self
    res = Ensemble()
    for e in self.__corps.keys(): res.ajoute(e)
    for e in autre.__corps.keys(): res.ajoute(e)
    return res

  (...)
```

Premier exemple : un type ensemble

```
  def intersect(self, autre):
    if not isinstance(autre, Ensemble): return None
    res = Ensemble()
    for e in self.__corps.keys():
       if e in autre.__corps.keys():
         res.ajoute(e)
    return res

  def diff(self, autre):
    if not isinstance(autre, Ensemble): return self
    res = Ensemble()
    for e in self.__corps.keys():
       if e not in autre.__corps.keys():
         res.ajoute(e)
    return res

if __name__ == '__main__':
  ens = Ensemble(12, 2, 1, 3)
  print(ens)                          # 1, 2, 3, 12
  ens.ajoute(30)
  print(ens)                          # 1, 2, 3, 12, 30
  ens.supprime(100)
  ens.supprime(30)
  print(ens)                          # 1, 2, 3, 12
  assert ens.contient(3)
  assert not ens.contient(42)
  ens2 = Ensemble(3, 2, 100, 34)
  print(ens2)                         # 2, 3, 100, 34
  print(ens.union(ens2))              # 1, 2, 3, 100, 12, 34
  print(ens.intersect(ens2))          # 2, 3
  print(ens.diff(ens2))               # 1, 12
```


Premier exemple : un type ensemble

Ça marche, mais c'est assez « Java-esque »... Voici une nouvelle version, plus « pythonique » :

```
class Ensemble:

  def __init__(self, *liste):
     self.__corps = {}
     for e in liste: self.__corps[e] = True

  def __iadd__(self, elt):
     self.__corps[elt] = True
     return self

  def __str__(self):
    s = ""
    for e in sorted(self.__corps.keys()): s += str(e) + ", "
    return s[:-2]

  def __or__(self, autre):
    if not isinstance(autre, Ensemble): return self
    res = Ensemble()
    for e in self: res += e   # Utilisation de __contains__ et de __iadd__
    for e in autre: res += e  # idem
    return res

(...)
```

Premier exemple : un type ensemble

```
def __and__(self, autre):
  if not isinstance(autre, Ensemble): return None
  res = Ensemble()
  for e in self:            # Utilisation de __contains__
     if e in autre:         # Utilisation de __contains__
       res += e             # Utilisation de __iadd__
  return res

def __sub__(self, autre):
  if not isinstance(autre, Ensemble): return self
  res = Ensemble()
  for e in self:
     if e not in autre:
       res += e
  return res

def __iter__(self):
  for e in sorted(self.__corps.keys()): yield e

def __getitem__(self, e):
  return self.__corps.get(e, False)

def __contains__(self, e):
  return self[e]

def __delitem__(self, e):
  self.__corps.pop(e, None)

def __len__(self):
  return len(self.__corps)
```


Premier exemple : un type ensemble

Et l'on peut maintenant écrire :

```
if __name__ == '__main__':
  ens = Ensemble(12, 2, 1, 3)
  print(ens)                                    # 1, 2, 3, 12
  ens += 30
  print(ens)                                    # 1, 2, 3, 12, 30
  del ens[100]
  del ens[30]
  print(ens)                                    # 1, 2, 3, 12
  assert ens[3]
  assert (3 in ens)
  assert not ens[42]
  assert (42 not in ens)
  ens2 = Ensemble(3, 2, 100, 34)
  print(ens2)                                   # 2, 3, 34, 100
  print(ens | ens2)                             # 1, 2, 3, 12, 34, 100
  print(ens & ens2)                             # 2, 3
  print(ens - ens2)                             # 1, 12
  for e in ens: print(e, end=" ")               # 1 2 3 12
  print()
```

Le module abc

- Le module *abc* (*Abstract Base Classes*) permet de définir des classes abstraites.
- Ce module fournit notamment la méta-classe *ABCMeta* qui permet de définir des classes abstraites.
- Il fournit également le décorateur *abstractmethod* pour définir des méthodes abstraites dans une classe dont la méta-classe est *ABCMeta* : la classe ne pourra être instanciée que si elle redéfinit toutes ses méthodes abstraites.
- Ce décorateur permet également de définir des méthodes de classes abstraites (on le combine avec *classmethod*) et des propriétés abstraites (on le combine avec *property*).
- Attention, en Python, le terme de *classe abstraite* n'est donc pas le même qu'en C++/Java/C# puisqu'une classe abstraite peut très bien ne pas avoir de méthode abstraites... (voir le premier exemple).
- Python fournit également les modules *numbers* et *collections* qui fournissent des classes abstraites pour les nombres et les collections (une classe qui hérite de *collections.Sequence* doit implémenter les méthodes *__getitem__* et *__len__*, par exemple, mais disposera automatiquement d'autres méthodes "mixins").

Exemple d'utilisation du module abc

```
from abc import ABCMeta, abstractmethod

class Figure(metaclass=ABCMeta):
  def __init__(self, nom):
    self._nom = nom

class FigureFermee(Figure):
  def __init__(self, nom):
    Figure.__init__(self, nom)

  @abstractmethod
  def surface(self):
    ...

class Rectangle(FigureFermee):
  def __init__(self, nom, coords, largeur, hauteur):
    FigureFermee.__init__(self, nom)
    self._coin_inf_gauche = coords
    self._largeur, self._hauteur = largeur, hauteur

  def surface(self):
    return self._largeur * self._hauteur

fig = Figure("abstraite")                          # Ok car pas de abstractmethod
fig_fermee = FigureFermee("toujours abstraite")    # TypeError: Can't instantiate abstract
                                                   # class FigureFermee with abstract methods surface
rect = Rectangle("carré", (1, 1), 10, 10)
rect.surface()                                     # 100
```

Exemple amélioré

```
class Figure(metaclass=ABCMeta):
  @abstractmethod
  def __init__(self, nom):
    self._nom = nom

  @property
  def nom(self):
    return self._nom

class FigureFermee(Figure):
  def __init__(self, nom):
    Figure.__init__(self, nom)

  @property
  @abstractmethod
  def surface(self):            # Toute figure fermée a une surface...
    ...

class Rectangle(FigureFermee):
  def __init__(self, nom, coords, largeur, hauteur):
    FigureFermee.__init__(self, nom)
    self._coin_inf_gauche = coords
    self._largeur, self._hauteur = largeur, hauteur

  @property
  def surface(self):          # Redéfinition de la propriété abstraite
    return self._largeur * self._hauteur

fig = Figure("abstraite")                          # TypeError: Can't instantiate abstract class...
rect = Rectangle("carré", (1, 1), 10, 10)
rect.surface                                       # 100
rect.nom                                           # 'carré'
```


Sérialisation et désérialisation

- La sérialisation consiste à transformer une structure de données en une suite d'octets afin de la stocker sur disque ou de l'envoyer sur le réseau. La désérialisation est l'opération inverse.
- Python dispose de plusieurs modules de sérialisation. Le plus utilisé est *pickle*, qui sérialise dans un format binaire uniquement lisible par Python.
- On peut également installer le module *yaml* qui sauvegarde les données dans un format lisible proche de XML ou le module *json* qui sauvegarde au format JSON (*JavaScript Object Notation*).
- Avec *pickle* et *json*, la sérialisation consiste à appeler les méthodes *dump* (pour écrire dans un fichier) ou *dumps* (pour écrire une chaîne d'octets) et la désérialisation consiste à appeler les méthodes *load* ou *loads*.
- Avec *yaml*, les méthodes *dump* et *load* traitent à la fois les fichiers et les chaînes d'octets.

Exemples

- Avec *pickle* :

```
import pickle

e1 = Ensemble(...)

# Sérialisation de e1 dans un fichier
with open('ensemble.pickle', 'wb') as f:
   pickle.dump(e1,f)                    # ou : pickle.dump(e1, open('ensemble.pickle', 'wb'))

# Désérialisation
with open('ensemble.pickle', 'rb') as f:
   e1 = pickle.load(f)                  # ou : e1 = pickle.load(open('ensemble.pickle', 'rb'))
```

- Avec YAML (il faut installer *PyYAML*) :

```
import yaml

e1 = Ensemble(...)

# Sérialisation de e1 dans un fichier
with open('ensemble.yaml', 'w') as f:
   yaml.dump(e1,f)                   # ou yaml.dump(e1, open('ensemble.yaml', 'w'))

# Désérialisation
with open('ensemble.yaml', 'r') as f:
   e1 = yaml.load(f)                 # ou e1 = yaml.load( open('ensemble.yaml', 'r'))
```


Sérialisation d'un Ensemble

```
import pickle, yaml
(...)
s = pickle.dumps(ens)                       # Sérialise dans un tableau d'octets
print(s)                                    # b'\x80\x03censemble\nEnsemble\nq\x00)...
ens3 = pickle.loads(s)
print(ens3)                                 # 1, 2, 3, 12

with open('ensemble.pickle', 'wb') as f:
  pickle.dump(ens, f)                       # Sérialise dans un fichier binaire

with open('ensemble.pickle', 'rb') as f:
  ens3 = pickle.load(f)

s = yaml.dump(ens)                          # YAML n'a qu'une méthode dump
print(s)                                    # !!python/object:ensemble.Ensemble
                                            # _Ensemble__corps: {1: true, 2: true, 3: true, 12: true}
ens4 = yaml.load(s)
print(ens3)                                 # 1, 2, 3, 12

with open('ensemble.yaml', 'w') as f:
  yaml.dump(ens, f)                       # Sérialise dans un fichier texte

with open('ensemble.yaml', 'r') as f:
  ens4 = yaml.load(f)
```

Fichiers texte

- La fonction *open()* permet d'ouvrir un fichier en lecture (mode *"r"* par défaut), en écriture (mode *"w"*) ou en ajout (mode *"a"*).
- Le mode peut être suivi de la lettre *b* pour indiquer une ouverture en mode binaire.
- En règle générale, il est préférable de considérer le fichier ouvert comme un itérateur classique (on peut alors utiliser une instruction *for*).
- Les méthodes *read()* et *readline()* renvoient une *chaîne* classique. La méthode *readlines()* renvoie un *tableau de chaînes* et n'ajoute pas de retour à la ligne.
- En lecture, la fin de fichier est détectée lorsque *read()* ou *readline()* renvoient une chaîne vide.
- La méthode *write()* écrit une chaîne dans le fichier ouvert en écriture. Elle renvoie le nombre de caractères écrits.
- On ferme un fichier avec la méthode *close()*.
- Si l'on a utilisé la construction *with*, le fichier ouvert est automatiquement fermé à la sortie du bloc.

Fichiers texte

Exemples :

```
fd = open("monfichier.txt")     # Ouverture en lecture par défaut
for ligne in fd:
   # faire quelque chose avec ligne
fd.close()                      # Fermeture explicite

for ligne in open("autre.txt"):
   # faire quelque chose avec ligne
# Ici le fichier est implicitement fermé car on est sorti de sa portée

fd = open("monfichier.txt")
contenu_complet = fd.read()     # une chaîne contenant tout le fichier

fd.seek(0)                      # on revient au début
prem = fd.readline()            # une chaîne contenant la 1ere ligne
sec  = fd.readline()            # une chaîne contenant la 2e ligne

fd.seek(0)
tab = fd.readlines()            # un tableau de lignes
print(tab[0])                   # première ligne...
```

Fichiers texte

Exemples :

```
fd = open("monfichier.txt", "w")     # Ouverture en écriture

# Si monfichier.txt existait, il a donc été écrasé !

fd.write("coucou\n")
fd.close()

with open("monfichier.txt") as fd:   # utilisation d'un bloc with
   # ici, fd est ouvert en lecture
# Sortie du bloc : fd est automatiquement fermé...

with open("monfichier.txt", "a") as fd:
  fd.write("au revoir\n")            # Ajout d'une ligne à la fin du fichier

from urllib.request import urlopen   # Gestion des URL comme des fichiers
import codecs

for ligne in urlopen("http://www.monsite.fr"):
   # on récupère des octets... donc il faut les décoder pour les transformer en caractères UTF-8
   print(codecs.decode(ligne))
```


Expressions régulières

Expressions régulières

- Les expressions régulières sont gérées via le module *re*, qu'il faut donc importer.
- Bien qu'elle ne soit pas nécessaire, la méthode *regexp = re.compile(motif)* permet d'optimiser les recherches. Elle permet également de passer des options (*re.I* ou *re.IGNORECASE*, par exemple).
- Si le *motif* contient des caractères spéciaux, on peut utiliser une *chaîne brute* en la préfixant du caractère *r* : *r"ceci est une chaîne brute"* (fonctionne également avec *'*, *'''* et *"""*).
- La méthode *search* d'une expression régulière renvoie un objet « match » si une correspondance a été trouvée, *None* sinon.
- La méthode *match* renvoie un objet « match » si la chaîne passée en paramètre correspond exactement au motif, *None* sinon.
- La méthode *sub* renvoie une chaîne où la première occurrence du motif dans la chaîne initiale a été remplacée par une autre chaîne.
- La méthode *findall* renvoie la liste de toutes les chaînes correspondant au motif dans la chaîne. La méthode *finditer* fait de même, mais renvoie un itérateur. On peut paramètrer le début et la fin de la recherche dans la chaîne.
- Voir les documentations complètes de ces méthodes dans la documentation du module *re*...

Opérations sur un objet « match »

Un objet « match » renvoyé par les appels à *search* ou *match* dispose des méthodes suivantes :

- *group(grp)* renvoie une chaîne correspond au texte capturé par l'expression ou la chaîne capturée par le groupe passé en paramètre (le groupe 0 correspond à toute la capture). Si l'on a utilisé des groupes nommés, on peut passer les noms en paramètre.
- *groups* renvoie un tuple contenant toutes les groupes capturés, à partir du groupe 1.
- *groupdict* renvoie un dictionnaire de tous les groupes nommés qui ont été capturés. Les clés sont les noms des groupes.
- *start* et *end* renvoient l'indice de début et de fin de la capture.

Rappels :

- Un groupe est délimité entre parenthèses. Un groupe est nommé par *?P<nom>*, *(?P<groupe>\d+)*, par exemple.
- Le traitement d'un groupe nommé est plus lent que celui d'un groupe non nommé.
- Ce qui a été capturé par un groupe est représenté par *\i* pour le groupe *i* ou par *(?P=nom)* pour le groupe nommé *nom*.
- Si l'on ne veut pas mémoriser le groupe capturé, on utilise la notation `(?: ...)`.

Exemple

On veut déterminer si un nom de fichier entré au clavier est un nom de fichier DOS valide, sachant que :

- Les noms de fichiers DOS ne sont pas sensibles à la casse
- Ils sont de la forme 8.3 : un nom principal et une extension, qui est facultative.
- Le nom principal doit commencer par une lettre et contenir des lettres des chiffres ou des blancs soulignés. Les lettres accentuées ne sont pas reconnues.
- L'extension ne contient que des lettres ou des chiffres.

Quelle expression régulière permettra de déterminer si la saisie est correcte et permettra d'isoler les deux parties du nom (nom principal et extension) ?

- Nom principal : `[a-zA-Z][a-zA-Z0-9_]{0,7}`
- Extension : `[a-zA-Z0-9]{1,3}`
- Nom complet, avec trois groupes pour capturer le nom principal et l'extension facultative : `([a-zA-Z][a-zA-Z0-9_]{0,7})(\.([a-zA-Z0-9]{1,3}))?`
- Idem, mais avec groupes nommés : `(?P<nom>[a-zA-Z][a-zA-Z0-9_]{1,7})(\.(?P<ext>[a-zA-Z0-9]{1,3}))?`

Exemple

```
import re

nom_dos = re.compile(r"(?P<nom>[a-zA-Z][a-zA-Z0-9_]{1,7})(\.(?P<ext>[a-zA-Z0-9]{1,3}))?")
nom_fic = "blabla.txt"                  # ou nom_fic = input("Nom du fichier : ")
result = re.match(nom_dos, nom_fic)
if result is not None:
    print(result.groups())          # ('blabla', '.txt', 'txt')
    nom, extension = result.group("nom"), result.group("ext")
    print(nom, extension)           # blabla txt
    nom, extension = result.group(1), result.group(3)
    print(nom, extension)           # blabla txt
else:
    print(nom_fic, "n'est pas un nom de fichier DOS")

nouveau_nom = re.sub(nom_dos, r"blibli\2", nom_fic)
print(nouveau_nom)                  # blibli.txt

print(re.sub(r"(\w+)\s+(\w+)", r"\2 \1", "mots inversés"))    # inversés mots

while True:
  age = input("Entrez un âge : ")
  if re.match("\d+", age) is not None:    # Plus besoin de try: ...
     age = int(age)                       # Mais age pourrait être une valeur non correcte...
     break
```

Programmation d'interfaces graphiques

Interfaces graphiques avec Python

- Il existe de nombreux « toolkits » graphiques, disponibles sur tous les systèmes (plus ou moins facilement...) : *Tk*, *Gtk*, *Qt*, *wxWidgets*.
- Ces toolkits étant généralement écrits en C ou en C++, les différents langages utilisent des « wrappers » afin d'accéder à leurs API. Pour les utiliser avec Python, il faut que les bibliothèques correspondantes soient installées sur votre système.
- Python est généralement fourni avec le module *tkinter* (*Idle* est une application Python/tkinter). Il est assez simple à programmer mais son look est un peu « daté ».
- GTK a été initialement conçu pour le développement du logiciel *The Gimp*, il a évolué en GTK2 et sa version actuelle, GTK3, est le toolkit utilisé pour l'environnement Gnome3. Pour l'utiliser avec Python, il faut installer *pyGTK* (pour GTK2) ou *pyGObject* (pour GTK3). Pour faciliter la création des interfaces GTK, on dispose de l'outil *Glade*.
- QT de Trolltech (devenu Qt Software depuis son rachat par Nokia) est la bibliothèque sur laquelle repose l'environnement KDE. Il a évolué en QT4, puis QT5 (avec quelques problèmes de compatibilité). Pour l'utiliser avec Python, il faut installer le paquetage *PyQt4* de Riverbank (utilisation libre pour les projets GPL, sinon payant) ou *PySide* (libre). Il existe également un paquetage *PyQT5*. L'outil *QT Designer*, fourni avec la distribution QT permet de créer des interfaces graphiques et l'outil *pyuic4* permet de les convertir en code Python.
- Pour ce cours, nous avons choisi *PyQt4*.

Python et QT4

- Le développement d'une application QT4 en Python s'effectue en plusieurs étapes :
  1. Création de l'interface de l'application avec *QT Designer* afin d'obtenir un fichier *.ui* au format XML (donc exploitable par tout langage disposant d'un wrapper QT).
  2. Conversion de ce fichier *.ui* en code source Python par la commande *pyuic4* : le fichier obtenu contient une classe Python décrivant l'interface graphique de l'application.
  3. Écriture des classes « métier » de l'application, dont l'une (la classe « principale ») héritera de la classe de l'interface graphique.
  4. Écriture d'un programme principal qui consistera à instancier cette classe et à créer la boucle d'événement du toolkit.
- Remarque : la pratique consistant à mélanger le code de l'interface et le code métier est à proscrire (c'est malheureusement assez fréquent dans les solutions pourries qu'on trouve sur les forums pour apprentis programmeurs...)
- Bien que le fichier *.ui* ne soit plus nécessaire après la création de la classe Python, conservez-le si vous voulez ensuite modifier l'interface (il faudra alors regénérer la classe Python correspondante).
- Les étapes 1 et 2 sont triviales et seront présentées en TP.

Principe général de l'application

- Python autorisant l'héritage multiple, la classe principale peut dériver à la fois de la classe QT correspondant au type de la fenêtre de notre application (*QWidget*, par exemple) et de la classe de notre interface, créée par *pyuic4*.
- Le constructeur de cette classe appelera le constructeur du widget principal, mettra en place l'interface graphique puis connectera les différents widgets pour qu'ils répondent aux actions de l'utilisateur.
- Pour illustrer cette démarche, nous supposerons que nous avons créé une classe Python nommée *Ui_ToutEnMaj* (QT Designer crée cette classe en rajoutant *Ui_* devant le nom que l'on a donné à notre widget principal). Nous avons choisi *QWidget* comme widget de base dans QT Designer et nous avons appelé ce widget *ToutEnMaj*.
- Cette interface contient un label (appelé *label*), un champ de saisie (appelé *lineEdit*) et un bouton (appelé *pushButton*).
- Nous voulons que le contenu du champ de saisie passe de minuscules en majuscules (et inversement) à chaque clic du bouton.

Écriture de la classe principale

- La classe principale doit importer certains modules de *PyQt4*, ainsi que notre classe *Ui_ToutEnMaj* que nous avons sauvegardée dans le fichier *toutenmaj.py* :

```
from PyQt4 import QtGui
from toutenmaj import Ui_ToutEnMaj
```

- Elle doit dériver de *QWidget* et de *Ui_ToutEnMaj* et son constructeur doit appeler celui de *QWidget* :

```
class MajQt(QtGui.QWidget, Ui_ToutEnMaj):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
```

- Puis, le constructeur doit mettre en place notre interface graphique par un appel à la méthode *setupUi* de *Ui_ToutEnMaj* :

```
self.setupUi(self)
```

- Enfin, il connecte le clic du bouton à une méthode *on_clic* que nous écrirons ensuite :

```
self.pushButton.clicked.connect(self.on_clic)
```


Application principale (suite)

- Il reste à écrire la méthode *on_clic* :

```
def on_clic(self):
    self.lineEdit.setText(self.lineEdit.text().swapcase())
```

- Et le programme principal de l'application, qui met en place l'environnement QT, crée une instance de notre classe et lance l'application :

```
if __name__ == '__main__':
    import sys
    app = QtGui.QApplication(sys.argv)
    win = MajQt()
    win.show()
    sys.exit(app.exec_())
```

Slots automatiques

- Dans la terminologie QT, les widgets *émettent* des *signaux*. Par exemple, un `QPushButton` peut émettre les signaux *clicked*, *pressed*, *released*, etc. (voir l'éditeur de signaux dans QT Designer).
- Les signaux sont traités par des *slots* qui sont représentés par des fonctions qui exécuteront leur code lorsque le signal sera reçu. En PyQT, on *connecte* à un slot le signal d'un widget à l'aide de la méthode *connect* :

```
self.pushButton.clicked.connect(self.on_clic)
```

- À partir de QT4, cette connection peut être automatique : il suffit qu'une méthode soit décorée par *@QtCore.pyqtSlot()* et que son nom soit de la forme *on_widget_signal(self)* pour que cette méthode soit considérée comme le slot du signal indiqué pour le widget indiqué.

Slots automatiques

La classe principale de l'exemple précédent devient donc :

```
from PyQt4 import QtGui, QtCore
from toutenmaj import Ui_ToutEnMaj


class MajQt(QtGui.QWidget, Ui_ToutEnMaj):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.pushButton.setToolTip('Bascule entre minuscules et majuscules')

    @QtCore.pyqtSlot()
    def on_pushButton_clicked(self):
        self.lineEdit.setText(self.lineEdit.text().swapcase())

if __name__ == '__main__':
    import sys
    app = QtGui.QApplication(sys.argv)
    win = MajQt()
    win.show()
    sys.exit(app.exec_())
```

Gestion des menus

- La gestion des menus avec QT est assez simple : le widget principal doit être un *QMainWindow* (et non un *QForm)*.
- Essentiellement, un *QMainWindow* possède une barre de menu (un *QMenuBar*), dans laquelle on place des *QMenu*.
- Chaque *QMenu* comporte des *QAction* qui représentent soit des options du menu, soit des séparateurs.
- Chaque *Qaction* émet le signal *triggered* lorsqu'il est sélectionné, *hovered* lorsque le curseur s'attarde sur lui, etc.
- Un *QMainMenu* possède également un *QStatusBar* qui permet de gérer une barre de statut située en bas de la fenêtre.

Exemple de gestion de menus

```
from PyQt4 import QtGui, QtCore
import sys

from FichiersUI import Ui_MainWindow

class OpenFileQT(QtGui.QMainWindow, Ui_MainWindow):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.actionQuitter.triggered.connect(QtGui.qApp.quit)

    @QtCore.pyqtSlot()
    def on_actionOuvrir_triggered(self):
        nomfic = QtGui.QFileDialog.getOpenFileName(
                        self,
                        "Ouvrir un fichier",
                        QtCore.QDir.homePath(),
                        "Tous les fichiers (*);; Fichiers images (*.png *.jpg)"
                    )
        if nomfic:
            self.label.setPixmap(QtGui.QPixmap(nomfic))


if __name__=='__main__':
    app = QtGui.QApplication(sys.argv)
    win = OpenFileQT()
    win.show()
    app.exec_()
```

Pour en savoir plus sur PyQT

- La documentation de PyQT : *http ://pyqt.sourceforge.net/Docs/PyQt4/*
- Des questions et des réponses sur StackOverflow : *http ://stackoverflow.com/search ?q=pyqt4*
- La documentation officielle de QT : *http ://qt-project.org/doc/*
- Attention aux « solutions » trouvées sur les forums ! Vérifiez la version de Python (3.x et non 2.x), vérifiez la version de PyQT, vérifiez qu'elle utilise bien les dernières évolutions (notamment pour la gestion des signaux et des slots), vérifiez qu'elles ne mélangent pas le code de l'UI et le code métier.