Table des matières

À quoi peut servir Python ?

Python est un langage de programmation **interprété**, puissant, à la fois facile à apprendre et riche en possibilités.
Il est, en outre, très facile d'étendre les fonctionnalités existantes (**bibliothèques**).
Concrètement, voilà ce qu'on peut faire avec Python (Python 3 - depuis 13 février 2009) :

- de petits programmes très simples, appelés **scripts**, chargés d'une mission très précise sur votre ordinateur
- des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, …
- des projets très complexes, comme des **progiciels** (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Mots clès : and, del, from, none, true, as, elif, global, nonlocal, try, assert, else, if, not, while, break, except, import, or, with, class, false, in, pass, yield, continue, finally, is, raise, def, for, lambda, return

Informations utiles :

- les **indentations** sont essentielles pour Python ;
  un moyen pour l'interpréteur de savoir où se trouvent le début et la fin d'un bloc
- commentaire = **"#"** ;
- l'objet vide de Python = **"None"**
- types de **variables** :
  - int # no size limit
  - float
  - chaines de caractères
  - bool (True, False)
- **type**(nom_de_la_variable) => retourne le type de la variable
- a, b = b, a **# permutation**
- range**:**
  - **range**(0, 5) # renvoie la liste [0, 1, 2, 3, 4] # jusqu'à N-1
  - **range**(5) # renvoie la liste [0, 1, 2, 3, 4] # jusqu'à N-1
  - **range**(0, 5, 2) # renvoie la liste [0, 2, 4] # jusqu'à N-1, avance de +2
- affichage :
  - print(a) # affiche 'a' suivi par un retour à la ligne
  - print(a, end = ' ') # affiche 'a' suivi par un espace
  - print("a =", a, "et b =", b) # affiche 'a =... et b =...' suivi par un retour à la ligne
  - print("a ={} b={}".format(a,b) # affiche 'a =... et b =...' suivi par un retour à la ligne
- **lire stdin** : year = **input**("Specify year: ")

  Par défaut entrée lue en type chaine de caractères => faut le convertir au besoin : year=**int**(year)

  Lire une liste des entiers :

  ```
  items = [ int(x) for x in input().split() ]                          # 1-d array
  items = [ [ int(x) for x in input().split() ] for i in range(n) ]    # 2-d array with 'n' rows
  items = list( map( int, input().split() ) )                          # map → spécifique à Python2
  ```

- **eval(...)**

  ```
  eval("9 + 5")      # => 14

  x = 2
  eval("x + 3")      # => 5
  ```

• Operations mathématiques

```
a // b                                   # la partie entière d'une division
divmod(a, b)                            # le tuple avec le quotient et le reste de la divison a/b
pow(a, b)                               # equivalent avec a**b   # a^b
pow(a, b, m)                            # a^b mod m
print("{:.2f}".format(a))   ou   round(a, 2)    # afficher 2 décimales
```

• Conditions : (if, elif, else)

```
if <condition>:
```

```
if age >= 18:
        print("Vous êtes majeur.")
else:
        print("Vous êtes mineur.")
```

```
if age < 10:
        print("Age inférieure à 10.")
elif age < 18:
        print("Age supérieure à 10 et inferieure à 18.")
else:
        print("Age supérieure à 18.")
```

• Comparaisons : <, >, <=, >=, ==, != (réponse: True, False)

→ comparaisons avec plusieurs prédicats : **and**, **or**, **not**, **is**

• Boucles :

```
while <condition>:                      for <element> in <sequence>:
        # bloc while                            # bloc for
```

Exemples :

```
i=1
while i <= 10:
    print(i, "*", nb, "=", i * nb)
    i += 1
```

```
chaine = "Bonjour les ZER0S"
for lettre in chaine:
    if lettre in "AEIOUYaeiouy":
        print(lettre)     # voyelle
```

```
N = int(input())
for i in range(0,N): # returns a list [0,1,...,N-1]
    print i
```

break, continue

```
while 1:
    lettre = input("Tapez 'Q' pour quitter : ")
    if lettre == "Q":
        print("Fin de la boucle")
        break   # on arrete l'execution du while
```

```
while i < 20:
    if i % 3 == 0:
        i += 4
        continue   # on retourne au while
    print("La variable i =", i)
    i += 1
```

• Format

Méthode de la classe 'str' : str.format()

Arguments :

```
"Nom={0}".format(nom)                          # premier argument
"Nom={}".format(nom)                           # premier argument implicitement
"Nom={}, Prenom={}".format(nom, prenom)        # premiers deux arguments implicitement
"Nom={1}, Prenom={0}".format(prenom, nom)      # respecte l'ordre d'appel d'arguments
"Poids={0.poids}".format(person)               # attribut 'poids' de l'objet passé en premier argument
"Premier joueur={[0]}".format(joueurs)         # premier élément ([0]) de la liste passé comme premier argument
```

La forme générale :

```
format_spec     [ [fill] align] [sign] [#] [0] [width] [,] [.precision] [type]

[fill]          →       <char>
[align]         →       "<" | ">" | "=" | "^"
[sign]          →       "+" | "-" | " "
[precision]     →       <entier>
[type]          →       "b" | "c" | "d" | "e" | "E" | "f" |"F" | "g" | "n" |"o" | "s" | "x" | "X" | "%"

[:#]            →       associée à une valeur binaire, octale ou hexadécimale => préfixe '0b', '0o' ou '0x'
[:,]            →       ajoute des virgules pour représenter les milliers
[:<width>]      →       aligne le texte sur <width> caractères (ajoute par default des espaces devant la valeur)
[:0<width>]     →       ajoute ( <width> - len(x) )  zéros devant x (quand x est un nombre)
```

Options de conversion :

```
"Nom={0!s}".format(nom)                        # appelle str() sur le premier argument
"Personne={0!r}".format(person)                # appelle repr() sur le premier argument
"Personne={0!a}".format(char)                  # appelle ascii() sur le premier argument
```

Options d'alignement :

```
"Nom={:<30}".format(nom)                       # premier argument aligné à gauche sur 30 caractères
"Nom={:>30}".format(nom)                       # premier argument aligné à droite sur 30 caractères
"Nom={:^30}".format(nom)                       # premier argument centré sur 30 caractères
"Nom={:*^30}".format(nom)                      # premier argument centré sur 30 caractères *
"Age={:=3}".format(age)                        # premier argument entier aligné à droite sur 3 caractères
```

Options de signe :

```
"x={:+f}, y={:+f}".format(3.14, -3.14)         # affiche le "+" et le "-" devant les nombres passés en argument
"x={: f}, y={: f}".format(3.14, -3.14)         # affiche un espace devant les nombres positives
"x={:-f}, y={:-f}".format(3.14, -3.14)         # affiche le - devant les nombres négatifs; équivalent à {:f}
```

Options de séparation :

```
"{:,}".format(123456789)                       # affiche le "," comme séparateur de miles => 1,234,567,890
```

Options de précision :

```
"{:.2%}".format(14.5/20)                       # affiche le pourcentage à 2 décimales => 72.50%
"{:.2f}".format(14.5/20)                       # affiche le flottant à 2 décimales => 0.72
```

Options de type :

```
"{:c}".format(32746)                                    # convertir en char => '猪'
"{:e}".format(10000)                                    # notation exposant; e minuscule => '1.000000e+04'
"{:E}".format(10000)                                    # notation exposant; e majuscule => '1.000000E+04'
"{:x}".format(42)                                       # convertir en hexadécimale, minuscules => 2a
"{:X}".format(42)                                       # convertir en hexadécimale, majuscules => 2A
"{:n}".format(42.5700000)                               # convertir en nombre => 42.57
"{:f}".format(42)                                       # convertir en flottant => 42.57000000
"int={0:d}; hex={0:x}; oct={0:o}; bin={0:b}; ".format(42)    # => 'int=42;  hex=2a;  oct=52;  bin=101010'
"int={0:d}; hex={0:#x}; oct={0:#o}; bin={0:#b}; ".format(42) # => 'int=42;  hex=0x2a;  oct=0o52;  bin=0b101010'
"{:%Y-%m-%d %H:%M:%S}".format(datetime.datetime(2010, 7, 4, 12, 15, 58))  # => '2010-07-04 12:15:58'
```

Exercice : affiche la forme décimale, octale, hexadécimale et binaire du 'x' sur le même nb de caractères

```
print("{0:{width}d} {0:{width}o} {0:{width}X} {0:{width}b}".format(x, width=width))
```

• Fonctions

```
def table(nb, max=10):                                  # variable max initialisé avec une valeur par défaut
    """Fonction affichant la table de multiplication par nb de 1*nb à max*nb  (max >= 0)"""
    i = 0
    while i < max:                                      # tant que i est strictement inférieure à 'max'
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1                                          # on incrémente i de 1 à chaque tour de boucle
```

=> vous pouvez appeler la fonction de deux façons :
- soit en précisant le numéro de la table et le nombre maximum d'affichages,
- soit en ne précisant que le numéro de la table (table(7));
  dans ce dernier cas, *max* vaudra 10 par défaut

Docstrings :
- une chaîne de caractères placés juste en-dessous de la définition de la fonction
- cette chaîne est ce qu'on appelle une docstring que l'on pourrait traduire par **une chaîne d'aide**
- si vous tapez **help(**table**)**, c'est ce message que vous verrez apparaître

Exemple :

```
def fonc(a=1, b=2, c=3, d=4, e=5):
    print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)

fonc()                          # => a = 1 b = 2 c = 3 d = 4 e = 5
fonc(4)                         # => a = 4 b = 2 c = 3 d = 4 e = 5
fonc(b=8, d=5)                  # => a = 1 b = 8 c = 3 d = 5 e = 5
fonc(b=35, c=48, a=4, e=9)      # => a = 4 b = 35 c = 48 d = 4 e = 9
```

Attention !
En Python, **la signature** d'une fonction est tout simplement **son nom**. Vous ne pouvez définir deux fonctions du même nom (si vous le faites, l'ancienne est écrasée par la nouvelle).

Fonctions lambda
- extrêmement courtes : limitées à une seule instruction
- pour l'appeler il faut stocker la fonction lambda dans une variable, par une simple affectation

```
lambda x: x * x                 # fonction qui prend un paramètre et renvoie son carré
carre = lambda x: x * x
carre(5)
```

Les fonctions dont on ne connaît pas à l'avance le nombre de paramètres (sans noms)

```
def fonction_inconnue(*parametres):
    """Test d'une fonction pouvant être appelée avec un nombre variable de paramètres"""
    print("J'ai reçu : {}.".format(parametres))

fonction_inconnue()             # => J'ai reçu : ().
fonction_inconnue('a', 'e', 'f')    # => J'ai reçu : ('a', 'e', 'f').
fonction_inconnue(var, [4], "...")  # => J'ai reçu : (3.5, [4], '...').
```

Les fonctions dont on ne connaît pas à l'avance le nombre de paramètres (avec noms)

```
def fonction_inconnue(**parametres_nommes):     # **parametres_nommes = un dictionnaire
    """Test d'une fonction permettant de voir comment récupérer les paramètres nommés"""
    print("J'ai reçu en paramètres nommés : {}.".format(parametres_nommes))

fonction_inconnue()             # => J'ai reçu : ().
fonction_inconnue(p=4, j=8)     # => J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
```

• **Modules Python** (stockés dans un espace de noms)

= un fichier contenant des fonctions et des variables, ayant toutes un rapport entre elles

→ si l'on veut travailler avec les fonctionnalités prévues par le module,
il n'y a qu'à importer le module et utiliser ensuite toutes les fonctions et variables prévues

```
import math
math.sqrt(16)                    # le nom du module suivi d'un « . » puis du nom de la fonction
help("math")                     # plus d'info sur le module "math" : description, fonctions, ...;
                                   Q (revenir), enter (avancer par ligne), espace (avancer par page)

import math as mathematiques     # importer le module math et l'héberger dans
                                   l'espace de noms dénommé « mathematiques » au lieu de math

from math import fabs            # importer que la fonction voulu,
                                   au lieu d'importer tout le module
from math import *               # importer les fonctions et variables du module math   dans l'espace de noms
                                   principal, sans les emprisonner dans l'espace de noms "math"
fabs(-15)
```

• **Programme Python**

Linux : `#!/usr/bin/python3.4` # header for program**.py**

Encodage : `# -*-coding:ENCODAGE -*` # header for program**.py**

```
Linux :      # -*-coding:utf-8 -*
Windows :    # -*-coding:Latin-1 -*
```

File : multiply.py

```
#!/usr/bin/python3.4
# -*-coding:utf-8 -*
"""module multipli contenant la fonction table"""

def table(nb, max=10):
  """fonction affichant la table de multiplication par nb de 1 * nb jusqu'à max * nb"""
  i = 0
  while i < max:
      print(i + 1, "*", nb, "=", (i + 1) * nb)
      i += 1
```

File : test.py

```
#!/usr/bin/python3.4
# -*-coding:utf-8 -*
from multipli import *

table(3, 20)                     # test de la fonction table
```

Pour tester la fonction 'table' directement dans le module :

```
if __name__ == "__main__":
    table(4)

os.system("pause")       # ajoute une pause pour visualiser le résultat (sous windows)
```

• Packages Python

Servent à regrouper des modules
En pratique, les packages sont… des répertoires !
Dedans on peut retrouver des **répertoires** (d'autres **packages**) ou des **fichiers** (des **modules**)

Exemple :

```
import nom_bibliotheque
nom_bibliotheque.evenements                    # pointe vers le sous-package evenements
nom_bibliotheque.evenements.clavier            # pointe vers le module clavier
from nom_bibliotheque.objets import bouton
```

Le fichier d'initialisation "__init__.py" optionnel depuis la version 3.3 de Python.
Vous pouvez y mettre du code d'initialisation pour votre package; ce code d'initialisation est appelé quand vous importez votre package.

Utilisation :

Dans votre répertoire de code, créez un **dossier** "package".
Dedans, créez un **fichier** "fonctions.py" dans lequel vous recopierez votre **fonction** "table".
Pour importer la fonction table:

```
from package.fonctions import table
table(5)                                       # appel de la fonction table
# ou
import package.fonctions
package.fonctions.table(5)                     # appel de la fonction table
```

• Exceptions Python

Python permet de tester un extrait de code.
S'il ne renvoie aucune erreur, Python continue.
Sinon, on peut lui demander d'exécuter une autre action.

Syntaxe :

```
try:
   # bloc à essayer
except:
  # bloc qui sera exécuté en cas d'erreur
```

Type d'erreur sur le code " annee = int(annee)" ou "resultat = numerateur / denominateur"

ValueError : erreur de conversion.
NameError : l'une des variables numerateur ou denominateur n'a pas été définie
TypeError : l'une des variables numerateur ou denominateur ne peut diviser ou être divisée
ZeroDivisionError : si denominateur vaut 0

Exemples :

```
annee = input()
try:                                           # on essaye de convertir l'année en entier
    annee = int(annee)
except:                                        # intercepte n'importe quelle exception liée à cette instruction
    print("Erreur lors de la conversion de l'année.")
```

```
try:
    resultat = numerateur / denominateur
except NameError:
    print("Variable numerateur ou denominateur pas définie.")
except TypeError:
    print("Variable numerateur ou denominateur incompatible avec la division.")
except ZeroDivisionError:
    print("Variable denominateur égale à 0.")
```

```
...
except typeException as exceptionRetournee:                # exceptionRetournee = nom de variable
    print("Voici l'erreur", exceptionRetournee)
```

Il est conseillé de toujours préciser un type d'exception possible après l'*except*.
Vous pouvez faire des tests dans l'interpréteur pour reproduire l'exception que vous voulez traiter.

Un exemple plus complet :

```
try:
    # test d'instruction(s)
except TypeDInstruction:
    # traitement en cas d'erreur
else:
    # permet d'exécuter une action si aucune erreur ne survient dans le bloc
finally:
    # instruction(s) exécutée(s) qu'il y ait eu des erreurs ou non
```

Dans le bloc de l'instruction "except" on peut utiliser le mot-clé **"pass"** (ne rien faire en cas d'erreur)

Assert : tester une condition

```
annee = input("Saisissez une année supérieure à 0 :")
try:
    annee = int(annee)                          # conversion de l'année
    assert annee > 0                            # vérifier si une condition est respectée
except ValueError:
    print("Vous n'avez pas saisi un nombre.")
except AssertionError:
    print("L'année saisie est inférieure ou égale à 0.")
```

Raise : lever une exception

```
annee = input()                                 # l'utilisateur saisit l'année
try:
    annee = int(annee)                          # on tente de convertir l'année
    if annee<=0:
        raise ValueError("l'année saisie est négative ou nulle")
except ValueError:
    print("La valeur saisie est invalide (l'année est peut-être négative).")
```

Pour **éviter que le programme s'arrête** après avoir rencontré une erreur :

```
nombre_mise = -1
while nombre_mise < 0 or nombre_mise > 49:
    nombre_mise = input("Tapez le nombre misé (entre 0 et 49) : ")
    try:
        nombre_mise = int(nombre_mise)
    except ValueError:
        print("Vous n'avez pas saisi de nombre")
        nombre_mise = -1
        continue                # continue l'exécution de la boucle 'while'
```

```
try:
    nombre_mise = int(nombre_mise)
except ValueError:
    pass                # ne fait rien en cas d'erreur
```

• OOP : les chaines de caractères

Une **classe** est un **modèle** qui servira à construire un **objet** (un objet est issu d'une classe)
Dans la classe on va définir les fonctions propres à l'objet (appelées méthodes).
En Python, une chaine de caractères représente un objet de la classe **"str".**

```
chaine = "NE CRIE PAS SI FORT !"

chaine.strip(), .lstrip(), .rstrip()                    # supprime les espaces en début et/ou fin de la chaine
chaine.lower(), .upper(), .capitalize(), .swapcase()    # mettre en minuscule, majuscules, capitaliser, ...

chaine.center(width, char), .rjust(), .ljust()  # aligner le texte sur #width chars (en ajoutant des chars avant et après)

chaine.startswith("..."), .endswith(".txt")  # vérifie si le texte commence ou se finisse par ...   # => bool (True, False)

chaine.isalnum(), .isalpha(), .isdigit()        # vérifie si le texte contient que des caractères
                                                   alphanumériques, alphabétiques ou numériques  # => bool (True, False)
chaine.isprintable()                            # vérifie si le texte contient que des caractères valides

chain.find("a", 0, -1)          # l'indice de la première occurrence de "a"; positions optionnelles; -1 if not found
chain.rfind("a", 0, -1)         # l'indice de la dernière occurrence de "a"; positions optionnelles; -1 if not found
chain.index(...), .rindex(...)  # pareil que find, rfind; sauf qu'ils soulèvent une exception si l'attribut n'est pas trouvé

chaine.count("a", 0, -1)        # le nb d'occurrences de la lettre "a" dans le string chaine[0:-1]; positions optionnelles

chaine.replace("a","b", 3)      # remplace les 3 premières occurrences de la lettre "a" par "b"; nombre optionnel

chaine.split(del)               # retourne une liste d'éléments délimités par 'del'; 'del' par default = ' ', '\t', '\n'
```

Liste complète de méthodes dans la classe "str" : **help(str)**
Aucune des méthodes de chaînes **ne modifie l'objet d'origine** mais elles **renvoient un nouvel objet**.

```
chaine = str()                                  # crée une chaîne vide; pareil que chaine = ""
while chaine.lower() != "q":
   print("Tapez 'Q' pour quitter...")
   chaine = input()
```

```
prenom = "Paul"
nom = "Dupont"
age = 21
print("Je m'appelle {0} {1} et j'ai {2} ans".format(prenom, nom, age))      # format = méthode de la classe "str"
nChaine = "Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom, nom, age)  # plus utile dans ce cas

print("Cela s'est produit le {}, à {}.".format(date, heure))        # numéros optionnels, si appelés dans la bonne ordre

adresse = """                                   # nommer les variables à afficher; plus intuitif que d'utiliser leur indice
{no_rue}, {nom_rue}
 {code_postal} {nom_ville} ({pays})
""".format(no_rue=5, nom_rue="rue des Postes", code_postal=75003, nom_ville="Paris", pays="France")
```

```
chaine_complete = message + " " + prenom                # on utilise le symbole '+' pour concaténer deux chaînes
```

Python est un langage à **typage dynamique**, ce qui signifie qu'il identifie lui-même les types de données et que les variables peuvent changer de type au cours du programme.
Python est un langage **fortement typé**, ce qui veut dire qu'on ne peut pas ignorer les types de données.

```
age = 21
message = "J'ai " + age + " ans."           # erreur !!!
message = "J'ai " + str(age) + " ans."      # faut convertir l'entier en chaine de caractères avant la concaténation
```

```
chaine[2]                   # troisième lettre de la chaîne
chaine[-1]                  # dernière lettre de la chaîne
len(chaine)                 # taille de la chaine de caractères (=# de caractères de la chaine)
chaine[0:2]                 # on sélectionne les deux premières lettres; équivalent de "chaine[:2]" # equiv substr
chaine[2:len(chaine)]       # on sélectionne la chaîne sauf les deux premières lettres; équivalent de "chaine[2:]"
mot = "lac"
mot = "b" + mot[1:]         # on remplace la lettre "l" avec la lettre "b"
```

• OOP : les listes

Les **listes** sont des **séquences** :
- des nombres entiers, des flottants, des chaînes de caractères, ou des objets de différents types.

Contrairement à la classe **str**, la classe **list** vous permet de remplacer un élément par un autre.
Les listes sont des types dits **mutables**.

Initialisation :

```
ma_liste = list()                       # une liste vide
ma_liste = []                           # une liste vide
ma_liste = [1, 2, 3, 4, 5]              # une liste avec cinq objets
ma_liste = [0] * 10                     # une liste avec dix elements initialisés avec 0
ma_liste = [1, 3.5, "une chaine", []]   # une liste avec 4 objets : un entier, un flottant, une chaine et une liste
ma_liste = list(listeX)                 # copie des éléments de la liste 'listeX'
```

Accès :

```
ma_liste[0]                             # on accède au premier élément de la liste
ma_liste[1] = 'Z'                       # on remplace le deuxième élément de la liste par 'Z'
```

Recherche :

```
ma_liste.index("a")                     # première occurrence de élément 'a' dans la liste
ma_liste.count("a")                     # nombre d'occurrences de élément 'a' dans la liste
```

Mise-à-jour de la liste :

```
ma_liste.append(56)                     # on ajoute 56 à la fin de la liste <=> on modifie l'objet d'origine !
ma_liste.insert(2, 'c')                 # on insère 'c' à l'indice 2
ma_liste1.extend(ma_liste2)             # on insère ma_liste2 à la fin de ma_liste1
ma_liste1 += ma_liste2                  # identique à extend
del ma_liste[0]                         # on supprime l'élément à l'indice '0' de la liste; fonctionnalité générale
ma_liste.pop()                          # on supprime le dernier élément de la liste et retourne cet élément
ma_liste.pop(index)                     # on supprime l'élément à l'indice 'index' de la liste et retourne cet élément
ma_liste.remove(32)                     # on supprime la première occurrence de l'élément "32" (pas l'indice)
ma_liste.sort()                         # on tri les éléments de la liste
ma_liste.reverse()                      # on inverse l'ordre des éléments dans la liste; ↔ ma_liste[::-1]
```

Parcourir une liste :

```
i = 0
while i < len(ma_liste):
    print(ma_liste[i])
    i += 1
```

```
for elt in ma_liste:    # elt = éléments de ma_liste
    print(elt)
for i, elt in enumerate(ma_liste):
    print("À l'indice {} se trouve {}.".format(i, elt))
```

Convertir des chaines en listes (split) et des listes en chaines (join) :

```
ma_chaine = "Bonjour à tous"
ma_chaine.split()   # par défaut split par espaces, tabulations et sauts de ligne # délimiteur optionnel   # => ['Bonjour', 'à', 'tous']

ma_liste = ['Bonjour', 'à', 'tous']
" ".join(ma_liste)                  # => 'Bonjour à tous' # afficher une liste sans parenthèses et virgules
```

Les compréhensions des listes : Syntaxe = **[expression for element in liste if predicat]**

= Parcourir une liste en en renvoyant une seconde, modifiée ou filtrée

```
liste_origine = [0, 1, 2, 3, 4, 5]
[x * x for x in liste_origine]                  # mettre au carré tous les nombres de la liste => [0, 1, 4, 9, 16, 25]
[x for x in liste_origine if x % 2 == 0]        # filtrer la liste : nombres pairs => [0, 2, 4]
[x  - nRemove for x in liste_origine if x > remove]   # => [1,2] (nRemove = 3)
```

Exercice : lire une liste des entiers; afficher tous les entiers sur une seule ligne, délimités par un espace

```
myArray = [ int(x) for x in input().strip().split() ]
print( " ".join( map(str, myArray) ) )          # 'join' demande en paramètre une liste des 'str'
print( " ".join( [ str(x) for x in myArray ] ) )    # équivalent, sans 'map'
```

Exercice : trier une liste de fruits en fonction de la quantité de chaque fruit

```
inventaire = [ ("pommes", 22), ("melons", 4), ("poires", 18), ("fraises", 76), ("prunes", 51) ]
inventaire_inverse = [ (qF, nomF) for nomF,qF in inventaire]        # change le sens de l'inventaire: (quantité, nom)
inventaire_inverse.sort(reverse=True)                               # trier dans l'ordre décroissant l'inventaire inversé
inventaire = [(nomF, qF) for qF,nomF in inventaire_inverse]         # on reconstitue l'inventaire
sorted(inventaire, key=lambda amount: amount[1], reverse=True)      # retourne la liste trié
inventaire.sort(key=lambda amount: amount[1], reverse=True)         # modifie la liste d'origine
```

Exercice : affichage d'un tableau en une ligne

```
joueurs = [ (15, "delroth"),  (2, "zozor"),  (0, "vali") ]

joueurs.sort()                  # trie d'abord avec le premier élément (le score), puis le deuxième (ordre alphabétique)
print( '\n'.join( [ "%s a %d points" % (joueur, score) for score, joueur in joueurs ] ) )        # spécifique Python2
# vali a 0 points
# zozor a 2 points
# delroth a 15 points
print( '\n'.join( [ "{} a {} points".format(joueur, score) for score, joueur in joueurs ] ) )    # équivalent, avec format
```

Exercice : affiche toutes les coordonnées 3D possibles avec Xi+Yi+Zi = N (multiples boucles 'for')

```
print( [ [xi,yi,zi] for xi in range(x+1) for yi in range(y+1) for zi in range(z+1) if xi+yi+zi != n ] )
```

Exercice : affiche les palindromes des 'n' entiers

```
print("".join( [ str(x) for y in ( range(1,n+1), range(n-1,0,-1) ) for x in y ] )
```

• OOP : les tuples

Les tuples sont des séquences qu'on ne peut pas modifier une fois créés.
Un tuple se définit comme une liste, en utilisant comme délimiteur des () au lieu des [].

```
tuple_vide = ()
tuple_non_vide = (1,)                       # est équivalent à ci dessous
tuple_non_vide = 1,
tuple_avec_plusieurs_valeurs = (1, 2, 5)
```

• OOP : les ensembles (set)

Pour créer un ensemble, on utilise un objet de type ***set***.
Cet objet a plein de méthodes cool à utiliser pour gérer les opérations avec les autres ensembles.
On crée un objet de type set en lui passant en paramètre une liste d'éléments qui sont dans l'ensemble.
L'objet ainsi crée ne peut pas contenir deux valeurs identiques (elles sont supprimées).

```
ens = {1, 2, 3, 4, 5, 6, 2, 3, 1, 7, 4}          # ensemble crée directement => {1, 2, 3, 4, 5, 6, 7}
ens = set([1, 2, 3, 4, 5, 6, 2, 3, 1, 7, 4])     # ensemble crée à partir d'une liste => {1, 2, 3, 4, 5, 6, 7}

ens.add(9)                                       # ajoute un élément : un entier
ens.add((5,4))                                   # ajoute un élément : un tuple
ens.update( [9, 10] )                            # ajoute une liste des éléments
ens.update( {-2, 11} )                           # ajoute un set des éléments
ens.discard(2)                                   # supprime l'élément '2'; ne fait rien s'il n'est pas dans la liste
ens.remove(5)                                    # supprime l'élément '5'; soulève une exception s'il n'est pas dans la liste

ens1 = set([1, 2, 3, 4, 5, 6])
ens2 = set([2, 3, 4, 8, 9])
ens3 = ens2.copy()                               # une copie 'superficielle' de l'ensemble ens2
ens3.clear()                                     # vide le contenu de l'ensemble ens3

ens1 & ens2  ↔ ens1.intersection(ens2)           # set([2, 3, 4]); intersection
ens1 | ens2  ↔ ens1.union(ens2)                  # set([1, 2, 3, 4, 5, 6, 7, 8, 9]); les deux réunis
ens1 - ens2  ↔ ens1.difference(ens2)             # set([1, 5, 6]), on enlève les éléments de ens2
ens1 ^ ens2  ↔ ens1.symmetric_difference(ens2)   # set([1, 5, 6, 8, 9]), l'union moins les éléments communs

ens2.issubset(ens1)                              # False; ens1 ne contient pas tous les éléments de ens2
ens1.issuperset(ens2)                            # False; ens1 ne contient pas tous les éléments de ens2
ens1.isdisjoint(ens2)                            # False; leur intersection n'est pas nulle
```

Les fonctions map et filter (spécifiques à Python2)

Les **fonctions callback** sont des fonctions utilisées comme argument d'une autre fonction.
Dans la pratique, en Python, rien ne permet de différencier une fonction callback d'une autre fonction, car elles s'utilisent exactement de la même manière.

Exemple:

```
def mettre_au_carre(x):
    return x ** 2                                   # x ** 2 = x puissance 2

def appliquer_fonction(fonc, valeur):
    return fonc(valeur)                             # on utilise la fonction callback fonc avec comme paramètre valeur

print(appliquer_fonction(mettre_au_carre, 3))       # affiche 9, c'est à dire mettre_au_carre(3)
```

L'utilisation de ces fonctions callback pour les fonctions **map** et **filter** :

- Syntaxe : **map(callback, liste)**

La fonction map permet de transformer une liste via l'utilisation d'une fonction callback.
En Python3 map retourne un objet de type map; pour récupérer une liste, faut ensuite le convertir en liste.

```
def carre(x): return x ** 2
def pair(x): return not bool(x % 2)

print(list(map(carre, [1, 2, 3, 4, 5])))      # affiche [1, 4, 9, 16, 25], c'est à dire le carré de chaque élément

print(list(map(pair, [1, 2, 3, 4, 5])))       # affiche [False, True, False, True, False], c'est à dire si le nombre est pair

for ch in map(chr,[65,66,67,68]):
    print(ch)                                 # convert to char; prints "ABCD"
```

- Syntaxe: **filter(callback, liste)**

La fonction filter permet de supprimer des valeurs d'une liste via l'utilisation d'une fonction callback.
En Python3 filter retourne un objet de type filter; pour récupérer une liste, faut ensuite le convertir en liste.

```
def petit_carre(x): return x ** 2 < 16
def pair(x): return not bool(x % 2)

print(list(filter(carre, [1, 2, 3, 4, 5])))      # affiche [1, 2, 3], c'est à dire les nombres dont les carrés sont inférieurs à 16

print(list(filter(pair, [1, 2, 3, 4, 5])))       # affiche [2, 4], c'est à dire les nombres pairs de la liste

for x in filter(pair, [1, 2, 3, 4, 5]):
    print(x)                                     # affiche [2, 4]
```

La fonction zip (spécifique à Python2)

Permet de combiner plusieurs listes en une seule, de manière à rendre les itérations plus simples.
En Python3 zip retourne un objet de type itérateur; pour récupérer une liste, faut ensuite le convertir en liste.

```
for e1, e2 in zip( liste1, liste2 ):
    # actions en utilisant e1 et e2

new_list = list ( zip( ['a', 'b', 'c'], ['d', 'e', 'f'] ) )          # => [  ('a', 'd'),    ('b', 'e'),    ('c', 'f')  ]
```

• OOP : les dictionnaires

Le dictionnaire est un **objet conteneur associant des clés à des valeurs**, sans aucune structure ordonnée. Pour accéder aux objets contenus dans le dictionnaire, on n'utilise pas nécessairement des indices mais des **clés** qui peuvent être de bien des types distincts.

Créer un dictionnaire :

```
mon_dictionnaire = dict()
mon_dictionnaire = {}
mon_dictionnaire = {"chemise":3, "pantalon":6}          # on sépare les différents couples clé:valeur par une virgule
```

Mise-à-jour du dictionnare :

```
mon_dictionnaire["pseudo"] = "Prolixe"  # si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée
mon_dictionnaire["pseudo"] = "6pri1"    # si la clé existe, l'ancienne valeur est remplacée par la nouvelle
echiquier['a', 1] = "tour blanche"      # la clé est un tuple ('a', 1)
del mon_dictionnaire["pseudo"]          # supprime la clé "pseudo" du dictionnaire
```

Accès :

```
mon_dictionnaire["pseudo"]              # si la clé n'existe pas => exception de type KeyError
```

Méthodes :

```
mon_dictionnaire.items()                # retourne les tuples du dictionnaire
mon_dictionnaire.keys()                 # retourne les clés du dictionnaire
mon_dictionnaire.values()               # retourne les valeurs du dictionnaire (valeurs associés au clés)
mon_dictionnaire.pop("pseudo")          # supprime la clé "pseudo" et retourne la valeur supprimée
```

Parcourir un dictionnaire :

```
fruits = {"pommes":21, "melons":3, "poires":31}
for cle in fruits:                                  # on parcourt la liste des clés contenues dans le dictionnaire
        print(cle)
for cle in fruits.keys():                           # on parcourt la liste des clés contenues dans le dictionnaire
        print(cle)
for valeur in fruits.values():                      # on parcourt la liste des valeurs contenues dans le dictionnaire
        print(valeur)
for cle, valeur in fruits.items():                  # on parcourt la liste des clés et valeurs contenues dans le dictionnaire
        print("La clé {} contient la valeur {}.".format(cle, valeur))
```

Note : on peut stocker les références des fonctions dans un dictionnaire

```
def fete():
        print("C'est la fête.")
def oiseau():
        print("Fais comme l'oiseau...")
fonctions = {}
fonctions["fete"] = fete                # on ne met pas les parenthèses
fonctions["oiseau"] = oiseau
fonctions["oiseau"]()                   # on essaye de l'appeler => Fais comme l'oiseau...
```

Note : un dictionnaire sans valeurs est un **set**

```
mon_set = {'pseudo', 'mot de passe'}    # ne peut pas contenir deux valeurs identiques
```

• Les fichiers

Quand on lance l'interpréteur Python, on un **répertoire de travail courant**.
Vous pouvez l'afficher grâce à la fonction **os.getcwd**() (cwd = « current working directory »).
Pour changer le répertoire de travail courant : **os.chdir(**"C:/tests python"**)** (oubliez pas "import os")

Ouvrir et fermer un fichier :

```
mon_fichier = open("fichier.txt", "r")          # ouvrir un fichier en mode lecture; "r" = read
mon_fichier = open("fichier.txt", "w")          # ouvrir un fichier en mode écriture; "w" = write
mon_fichier = open("fichier.txt", "a")          # ouvrir un fichier en mode ajout; "a" = append
mon_fichier = open("fichier.txt", "wb")         # ouvrir un fichier en mode écriture binaire
mon_fichier.close()                             # fermer le fichier
```

Lire un fichier :

```
contenu = mon_fichier.read()                    # lire l'intégralité du fichier
```

Ecrire dans un fichier : la méthode write n'accepte en paramètre que des chaînes de caractères

```
contenu = mon_fichier.write("Premier test d'écriture dans un fichier via Python")  # renvoie le nombre de caractères
```

Le mot clé with : permet de créer un "context manager" qui vérifie que le fichier est ouvert et fermé

```
with open('file.txt', 'r') as mon_fichier:
    texte = mon_fichier.read()
```

Module pickle : enregistrer et récupérer des objets

```
import pickle
score = { "joueur 1": 5,  "joueur 2":35, "joueur 3":20, "joueur 4":2 }

with open('donnees', 'wb') as fichier:
    mon_pickler = pickle.Pickler(fichier)       # classe Pickler => pour l'enregistrement d'objets
    mon_pickler.dump(score)                     # sauvegarde l'objet 'score' dans le fichier binaire 'donnees'

with open('donnees', 'rb') as fichier:
    mon_depickler = pickle.Unpickler(fichier)   # classe Unpickler => pour l'enregistrement d'objets
    score_recupere = mon_depickler.load()       # récupère le premier objet contenu dans le fichier binaire 'donnees'
```

• Portée des variables et références

La portée c'est « quand et comment les variables sont-elles accessibles ? »
(quelles variables sont accessibles depuis le corps d'une fonction et de quelle façon)

L'espace local

Dans une fonction, quand vous faites référence à une variable :

- Python vérifie dans **l'espace local de la fonction**;
  cet espace contient les paramètres qui sont passés à la fonction et les variables définies dans son corps
- si la variable n'existe pas dans l'espace local de la fonction,
  Python va regarder dans l'espace local dans lequel la fonction a été appelée.

  - les **variables locales** définies **avant l'appel d'une fonction** seront accessibles, depuis le corps de la fonction, en lecture seule
  - une **variable locale** définie **dans une fonction** sera supprimée après l'exécution de cette fonction
  - l'espace local d'une fonction est détruit après l'appel de la fonction
  - une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace
  - on peut modifier le contenu d'une variable en faisant appel à ses méthodes

```
def ajouter(liste, valeur_a_ajouter):
    """Cette fonction insère à la fin de la liste la valeur que l'on veut ajouter"""
    liste.append(valeur_a_ajouter)
```

Références :

```
ma_liste1 = [1, 2, 3]
ma_liste2 = ma_liste1          # ma_liste1 et ma_liste2 contiennent une référence vers le même objet
ma_liste2.append(4)            # si on modifie l'objet depuis 1 de ces variables, le changement sera visible depuis les 2
print(ma_liste2)               # => [1, 2, 3, 4]
print(ma_liste1)               # => [1, 2, 3, 4]
```

Créer un nouvel objet depuis un autre :

```
ma_liste1 = [1, 2, 3]
ma_liste2 = list(ma_liste1)    # cela revient à copier le contenu de ma_liste1
ma_liste2.append(4)
print(ma_liste2)               # => [1, 2, 3, 4]
print(ma_liste1)               # => [1, 2, 3]
```

Comparer les objets :

```
ma_liste1 = [1, 2]
ma_liste2 = [1, 2]
ma_liste1 == ma_liste2         # on compare le contenu des listes => True
ma_liste1 is ma_liste2         # on compare leur référence => False
id(ma_liste1)                  # la position de l'objet ma_liste1 dans la mémoire Python => 56171784
id(ma_liste2)                  # la position de l'objet ma_liste2 dans la mémoire Python => 56169544
```

Variables globales :

```
i = 4                          # une variable, nommée i, contenant un entier
def inc_i():
    """Fonction chargée d'incrémenter i de 1"""
    global i                   # Python recherche i en dehors de l'espace local de la fonction
    i += 1
```

En précisant **global i**, Python permet l'accès en lecture et en écriture à cette variable (préalablement crée), ce qui signifie que vous pouvez changer sa valeur par affectation.

• OOP : les classes

Une classe est un modèle suivant lequel on va créer des objets. C'est dans la classe que nous allons définir nos **méthodes** et **attributs** (variables contenues dans notre objet).

Convention de nommage : **C**amel**C**ase (mettre en majuscule chaque lettre débutant un mot)

Exemple : une classe pour définir une personne

```
class Personne :                                    # définition de la classe
    """Classe définissant une personnes caractérisée par son : nom, prénom, âge et lieu de résidence"""

    nb_personnes = 0                                # attribut de classe

    def __init__(self, nom, prenom, age, residence):   # la méthode constructeur (avec ou sans paramètres)
        self.nom = nom
        self.prenom = prenom
        self.age = age
        self.lieu_residence = residence
        Personne.nb_personnes += 1                  # on préfixe le nom de l'attribut de classe par le nom de la classe
```

Votre paramètre ***self***, c'est l'objet qui appelle la méthode.
Cela peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques.

Exemple : méthodes de classe et méthodes statiques

```
class Compteur :                                    # définition de la classe
    """Classe qui possède un attribut de classe qui s'incrémente à chaque fois que l'on crée un objet de ce type"""
    objets_crees = 0                                # attribut de classe
    def __init__(self):                             # la méthode constructeur (sans paramètres)
        Compteur. objets_crees += 1
    def combien(cls):                               # methode de classe, prend un premier paramètre "cls"
        """Méthode de classe affichant combien d'objets ont été créés"""
        print("Jusqu'à présent, {} objets ont été créés.".format(cls.objets_crees))
    combien = classmethod(combien)                  # on utilise ensuite une fonction built-in de Python pour lui faire
                                                    # comprendre qu'il s'agit d'une méthode de classe,
                                                    # pas d'une méthode d'instance.

    def afficher():                                 # methode statique, ne prend aucun premier paramètre
        """Fonction chargée d'afficher quelque chose"""
        print("On affiche la même chose.")
        print("peu importe les données de l'objet ou de la classe.")
    afficher = staticmethod(afficher)
```

L'encapsulation

L'encapsulation est un principe qui consiste **à cacher ou protéger** certaines données de notre objet.
Nos attributs d'objets ne doivent pas être accessibles depuis l'extérieur de la classe.
Autrement dit, vous n'avez pas le droit de faire, depuis l'extérieur de la classe, monObjet.monAttribut.

=> On va définir des méthodes un peu particulières, appelées des **accesseurs et mutateurs**.
Les accesseurs donnent accès à l'attribut (**monObjet.get_monAttribut**()).
Les mutateurs permettent de le modifier (**monObjet.set_monAttribut**(valeur)).

En Python, il n'y a pas d'attribut privé. Tout est public.
Cela signifie que si vous voulez modifier un attribut depuis l'extérieur de la classe, vous le pouvez.

Les **propriétés** sont un moyen transparent de manipuler des attributs d'objet.
Elles permettent de dire à Python : « Quand un utilisateur souhaite modifier cet attribut, fais cela ».
De cette façon, on peut :
- rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe,
- ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable,
- ou faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.

Une propriété attend quatre paramètres, en ordre, tous optionnels : la méthode donnant accès à l'attribut (get), la méthode modifiant l'attribut (set), la méthode appelée quand on souhaite supprimer l'attribut (del) et la méthode appelée quand on demande de l'aide sur l'attribut (help)

```
class Personne:
    """Classe définissant une personne caractérisée par son : nom, prénom, âge, lieu de résidence"""

    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self._residence = "Paris"          # notez le souligné _ devant le nom;
                                           # la convention veut qu'on n'accède pas à un attribut commençant par un _
    def _get_residence(self, residence):
        """Méthode appelée quand on souhaitera accéder en lecture à l'attribut '_residence' """
        print("On accède à l'attribut 'residence' !")
        return self._residence

    def _set_residence(self, new_residence):
        """Méthode appelée quand on souhaite modifier le lieu de '_résidence' """
        print("Attention, il semble que {} déménage à {}.".format(self.prenom, new_residence))
        self._residence = new_residence

    # on va dire à Python que notre attribut 'residence' pointe vers une propriété
    residence = property(_get_residence, _set_residence)                    # définition d'une propriété
```

Les méthodes spéciales

Les méthodes spéciales sont des **méthodes d'instance** que Python reconnaît et sait utiliser, dans certains contextes. Elles peuvent servir à indiquer à Python ce qu'il doit faire quand il se retrouve devant une expression comme `mon_objet1 + mon_objet2`, voire `mon_objet[indice]`. Et, encore plus fort, elles contrôlent la façon dont un objet se crée, ainsi que l'accès à ses attributs.

Le nom d'une méthode spéciale a la forme : __methodeSpeciale__.

Exemples :

- constructeur : __init__ ; destructeur : __del__ ; affichage : __repr__ & __str__

```
def __repr__(self):
    """Quoi afficher quand on entre notre objet dans l'interpréteur"""
    return "Personne: nom({}), prénom({}), âge({})".format(self.nom, self.prenom, self.age)
```

La méthode __repr__ ne prend aucun paramètre et renvoie une chaîne de caractères :
=> la chaîne à afficher quand on entre l'objet directement dans l'interpréteur.

```
def __str__(self):
    """Méthode permettant d'afficher plus joliment notre objet"""
    return "{} {}, âgé de {} ans".format(self.prenom, self.nom, self.age)
```

La méthode __str__ est spécialement utilisée pour afficher l'objet avec **print**. Par défaut, si aucune méthode *str* n'est définie, Python appelle la méthode *repr* de l'objet. La méthode *str* est également appelée si vous désirez convertir votre objet en chaîne (avec *str()*).

- accès aux attributs : __getattr__ & __setattr__ & __delattr__

```
def __getattr__(self, nom):
    """Si Python ne trouve pas l'attribut nommé nom, il appelle cette méthode. On affiche une alerte"""
    print("Alerte ! Il n'y a pas d'attribut {} ici !".format(nom))
```

La méthode __getattr__ permet de définir une méthode d'accès à nos attributs.
Cette méthode est appelée quand vous tapez objet.attribut (pour y accéder).
Python recherche l'attribut et, s'il ne le trouve pas dans l'objet et si une méthode *getattr* existe, il va l'appeler avec un paramètre, le nom de l'attribut recherché sous forme d'une chaîne de caractères.

```
def __setattr__(self, nomAttr, valAttr):
    """Méthode appelée quand on fait objet.nomAttr = valAttr. On se charge d'enregistrer l'objet"""
    object.__setattr__(self, nomAttr, valAttr)          # heritage de la classe 'object'
    self.enregistrer()                                  # erreur ?
```

Cette méthode définit l'accès à un attribut destiné à être modifié.
Si vous écrivez objet.nomAttribut = nouvelleValeur,
la méthode spéciale __setattr__ sera appelée ainsi : objet.__setattr__("nomAttribut", nouvelleValeur)

```
def __delattr__(self, nom_attr):
    """On ne peut supprimer d'attribut, on lève l'exception AttributeError"""
    raise AttributeError("Vous ne pouvez supprimer aucun attribut de cette classe")
    # ou, pour réellement supprimer l'attribut :  object.__delattr__(self, nomAttr)
```

Cette méthode spéciale est appelée quand on souhaite supprimer un attribut de l'objet.
Elle prend en paramètre (outre self) le nom de l'attribut que l'on souhaite supprimer.

Des fonctions ayant un comportement similaire

```
objet = MaClasse()              # on crée une instance de notre classe
getattr(objet, "nom")           # semblable à objet.nom
setattr(objet, "nom", val)      # équivalent à  objet.nom = val   ou  objet.__setattr__("nom", val)
delattr(objet, "nom")           # équivalent à  del objet.nom     ou  objet.__delattr__("nom")
hasattr(objet, "nom")           # renvoie True si l'attribut "nom" existe, False sinon
```

Les méthodes de conteneurs

Les **objets conteneurs** (les chaînes de caractères, les listes et les dictionnaires, entre autres) ont tous un point commun : ils contiennent d'autres objets, auxquels on peut accéder grâce à l'opérateur [ ].

```
class ZDict:
    """Classe enveloppe d'un dictionnaire"""

    def __init__(self):
        """Notre classe n'accepte aucun paramètre"""
        self._dictionnaire = {}

    def __getitem__(self, index):
        """Méthode appelée quand on fait objet[index].
        Elle redirige vers self._dictionnaire[index]"""
        return self._dictionnaire[index]

    def __setitem__(self, index, valeur):
        """Méthode appelée pour 'objet[index] = valeur' """
        self._dictionnaire[index] = valeur

    def __delitem__(self, index):
        """Méthode appelée quand on écrit del objet[index] """
        del self._dictionnaire[index]

    def __contains__(self, valeur):
        """Méthode appelée quand on écrit 'value in objet' """
        return valeur in self._dictionnaire

    def __len__(self):
        """Méthode appelée quand on écrit 'len(objet)' """
        return len(self._dictionnaire)
```

Les méthodes mathématiques

Les méthodes spéciales permettant la surcharge d'opérateurs mathématiques, comme +, -, *

```
class Duree:
    """Classe contenant des durées sous la forme d'un nombre de minutes et de secondes"""

    def __init__(self, min=0, sec=0):
        """Constructeur de la classe"""
        self.min = min          # nombre de minutes
        self.sec = sec          # nombre de secondes
    def __str__(self):
        """Affichage un peu plus joli de nos objets"""
        return "{0:02}:{1:02}".format(self.min, self.sec)

    def __add__(self, objectToAdd):
        """L'objet à ajouter est un entier, le nombre de secondes (opération classe + entier )"""
        nouveleDuree = Duree()
        nouvelleDuree.min = self.min                    # on va copier self dans l'objet créé ( => la même durée)
        nouvelleDuree.sec  = self.sec
        nouvelleDuree.sec += objectToAdd                # on ajoute la durée
        if nouvelleDuree.sec >= 60:                     # si le nombre de secondes >= 60
            nouvelleDuree.min += nouvelle_duree.sec // 60
            nouvelleDuree.sec = nouvelle_duree.sec % 60
        return nouvelleDuree                            # on renvoie la nouvelle durée
```

```
def __radd__(self, objectToAdd):
    """Cette méthode est appelée si on écrit 4 + objet et que le premier objet (4 dans cet exemple) ne sait pas comment
    ajouter le second. On se contente de rediriger sur __add__ puisque, ici, cela revient au même :
    l'opération doit avoir le même résultat, posée dans un sens ou dans l'autre"""
    return self + objectToAdd

def __iadd__(self, objet_a_ajouter):
    """L'objet à ajouter est un entier, le nombre de secondes (opération classe += entier )"""
    self.sec += objet_a_ajouter                    # ici on travaille directement sur self; on ajoute la durée

    if self.sec >= 60:                             # si le nombre de secondes >= 60
        self.min += self.sec // 60
        self.sec = self.sec % 60

    return self                                    # on renvoie self
```

D'autres méthodes : __sub__ (-), __mul__ (*), __truediv__ (/), __floordiv__ (//), __mod__ (%), __pow__ (**)

Les méthodes de comparaison

Pour finir, nous allons voir la surcharge des opérateurs de comparaison: ==, !=, <, >, <=, >=.
Ces méthodes sont donc appelées si vous tentez de comparer deux objets entre eux.

Les méthodes : __eq__ (==), __ne__ (!=), __gt__ (>), __ge__ (>=), __lt__ (<), __le__ (<=)

```
def __eq__(self, autreDuree):
    """Test si self et autreDuree sont égales"""
    return self.sec == autreDuree.sec and self.min == autreDuree.min

def __gt__(self, autreDuree):
    """Test si self > autreDuree"""
    nbSec1 = self.sec + self.min * 60
    nbSec2 = autreDuree.sec + autreDuree.min * 60
    return nbSec1 > nbSec
```

Des méthodes spéciales utiles à pickle

Deux méthodes utilisées pour influencer la façon dont nos objets sont enregistrés dans des fichiers.

La méthode __getstate__ est appelée au moment de sérialiser l'objet.
Si aucune méthode __getstate__ n'est définie, pickle enregistre le dictionnaire des attributs de l'objet à enregistrer (contenu dans objet.__dict__). Sinon, pickle enregistre dans le fichier la valeur renvoyée par __getstate__ (généralement, un dictionnaire d'attributs modifié).

```
class Temp:
    """Classe contenant plusieurs attributs, dont un temporaire"""

    def __init__(self):
        """Constructeur de notre objet"""
        self.attribut1 = "une valeur"
        self.attribut2 = "une autre valeur"
        self.attributTemporaire = 5

    def __getstate__(self):
        """Renvoie le dictionnaire d'attributs à sérialiser"""
        newDict = dict(self.__dict__)
        newDict ["attributTemporaire"] = 0
        return newDict
```

La méthode __setstate__ est appelée au moment de désérialiser l'objet.
Si vous récupérez un objet à partir d'un fichier sérialisé, __setstate__ sera appelée après la récupération du dictionnaire des attributs.

Voici l'exécution que l'on va observer derrière **unpickler.load()** :
- l'objet Unpickler lit le fichier
- il récupère le dictionnaire des attributs
- ce dictionnaire récupéré est envoyé à la méthode __setstate__ si elle existe; si elle n'existe pas, Python écrit l'attribut __dict__ de l'objet en y plaçant ce dictionnaire récupéré

```
    def __setstate__(self, newDict):
        """Méthode appelée lors de la désérialisation de l'objet"""
        newDict ["attribut_temporaire"] = 0          # ici on modifie le dictionnaire d'attributs après la désérialisation
        self.__dict__ = newDict
```

• L'héritage

L'héritage est une fonctionnalité objet qui permet de déclarer que **telle classe sera elle-même modelée sur une autre classe**, qu'on appelle la classe parente, ou la classe mère.

Concrètement, si une classe ***b*** hérite de la classe ***a***, les objets créés sur le modèle de la classe ***b*** auront accès aux méthodes et attributs de la classe ***a***. En plus, la classe ***b*** dans va pouvoir définir d'autres méthodes et attributs dans son corps (qui lui seront donc propres), ou même redéfinir les méthodes de la classe mère.

L'héritage simple

```
class A:
    """Classe A, pour illustrer
    notre exemple d'héritage"""
    pass     # on laisse la définition vide
```

```
class B(A):          # syntaxe class MaClasse(MaClasseMere):
    """Classe B, qui hérite de A;
    elle reprend les mêmes méthodes et attributs """
    pass
```

Si vous faites **objetB.maMethode**(),
- Python va d'abord chercher la méthode **maMethode** dans la **classe B** dont l'objet est directement issu
- s'il ne trouve pas, il va chercher récursivement dans les classes dont hérite B, c'est-à-dire **A** (dans cet ex)

Si aucune méthode n'a été redéfinie dans la classe, on cherche dans la classe mère. On peut ainsi redéfinir une certaine méthode dans une classe et laisser d'autres directement hériter de la classe mère.

```
class Personne:
    """Classe représentant une personne"""

    def __init__(self, nom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = "Martin"

    def __str__(self):
        """Méthode appelée lors d'une conversion
        de l'objet en chaîne"""
        return "{0} {1}".format(self.prenom, self.nom)
```

```
class AgentSpecial(Personne):
    """Classe définissant un agent spécial; hérite la classe Personne"""

    def __init__(self, nom, matricule):
        """Un agent se définit par son nom et son matricule"""
        Personne.__init__(self, nom)  # constructeur Personne
        self.matricule = matricule

    def __str__(self):
        """Méthode appelée lors d'une conversion objet-> chaîne"""
        return "Agent {0}, matricule {1}".format(\
                                    self.nom,    self.matricule)
```

Deux fonctions très pratiques :

- **issubclass** : vérifie si une **classe** est une sous-classe d'une autre classe (=> True / False)

```
issubclass(AgentSpecial, Personne)     # AgentSpecial hérite de Personne => True
issubclass(AgentSpecial, object)       # AgentSpecial hérite de 'object'  => True
issubclass(Personne, AgentSpecial)     # Personne n'hérite pas d'AgentSpecial => False
```

- **isinstance** : permet de savoir si un **objet** est issu d'une classe ou de ses classes filles

```
agent = AgentSpecial("Fisher", "18327-121")
isinstance(agent, AgentSpecial)        # agent est une instance d'AgentSpecial => True
isinstance(agent, Personne)            # agent est une instance héritée de Personne => True
```

L'héritage multiple (hériter de plusieurs classes)

```
class MaClasseHeritee(MaClasseMere1, MaClasseMere2):
```

Pour rechercher une méthode, c'est l'ordre de définition des classes mères qui importe.

On va chercher la méthode dans les classes mères de gauche à droite. Si on ne trouve pas la méthode dans une classe mère donnée, on remonte dans ses classes mères, et ainsi de suite.

Création d'exceptions personnalisées :

```
class MonException(Exception):
    """Exception levée dans un certain contexte... qui reste à définir"""

    def __init__(self, message):
        """On se contente de stocker le message d'erreur"""
        self.message = message

    def __str__(self):
        """On renvoie le message"""
        return self.message
```

• Les métaclasses

== comment générer des classes à partir… d'autres classes !

Création dynamique d'une classe :

```
def creer_personne(personne, nom, prenom):
    """La fonction qui jouera le rôle de constructeur pour notre classe Personne.
    Elle prend en paramètre, outre la personne :    nom -- son nom;  prenom -- son prenom"""
    personne.nom = nom
    personne.prenom = prenom
    personne.age = 21
    personne.lieu_residence = "Lyon"

def presenter_personne(personne):
    """Fonction présentant la personne; elle affiche son prénom et son nom"""
    print("{} {}".format(personne.prenom, personne.nom))

methodes = {                                    # dictionnaire des méthodes
    "__init__": creer_personne,
    "presenter": presenter_personne,
}

Personne = type("Personne", (), methodes)       # création dynamique de la classe;
                                                # type est la métaclasse de toutes les classes par défaut
```

Le processus d'**instanciation d'un objet** est assuré par deux méthodes, __new__ et __init__.
- __new__ est chargée de la **création de l'objet** et prend en premier paramètre sa classe
- __init__ est chargée de l'initialisation des attributs de l'objet et prend en premier paramètre l'objet précédemment créé par __new__

Les classes étant des objets, elles sont toutes modelées sur une classe appelée **métaclasse**.

À moins d'être explicitement modifiée, la métaclasse de toutes les classes est ***type***.

On peut utiliser ***type*** pour créer des classes dynamiquement.

On peut faire hériter une classe de ***type*** pour **créer une nouvelle métaclasse**.

Dans le corps d'une classe, pour spécifier sa métaclasse, on exploite la syntaxe suivante :

class MaClasse(**metaclass=NomDeLaMetaClasse**):

• Les classes singleton

Une classe dite singleton est une classe qui ne peut être instanciée qu'une fois.
Autrement dit, on ne peut créer qu'un seul objet de cette classe.
Ceci est très facile à modéliser grâce à des décorateurs.

```
def singleton(classe_definie):
    instances = {}                                  # dictionnaire de nos instances singletons
    def get_instance():
        if classe_definie not in instances:
            instances[classe_definie] = classe_definie()    # on crée notre premier objet de classe_definie
        return instances[classe_definie]
    return get_instance

@singleton                                          a = Test()
class Test:                                         b = Test()      # a et b pointent vers le même objet
    pass                                            a is b          # => True
```

• Le tri en Python

Pour trier une séquence de données, Python nous propose **deux méthodes** :

- La première est une méthode de liste : **sort**
  Elle travaille sur la liste-même et change donc son ordre, si c'est nécessaire.
- La seconde est la fonction **sorted**()
  Cette fonction travaille sur n'importe quel type de séquence; une importante différence avec la méthode list.sort est qu'elle ne modifie pas l'objet d'origine, mais en retourne un nouveau.

Ordonner une liste des chaines de caractères :

```
prenoms = ["Jacques", "Laure", "André", "Victoire", "Albert", "Sophie"]
prenoms.sort()                  # => ['Albert', 'André', 'Jacques', 'Laure', 'Sophie', 'Victoire']; la liste est modifiée
sorted(prenoms)                 # => ['Albert', 'André', 'Jacques', 'Laure', 'Sophie', 'Victoire']; ne modifie pas la liste
```

Pour Python, la méthode de tri dépend du type des éléments que la séquence contient.
On lui demande de trier une liste de nombres (type int) et Python trie du plus petit au plus grand.
On lui demande de trier une liste de chaines (type str) et Python trie par ordre alphabétique.

Les deux méthodes de tri ont un paramètre optionnel, appelé **key**. Cet argument attend une fonction.
Pour préciser la méthode de tri, il nous faut donc une fonction qui prenne en paramètre un élément de la liste à trier et retourne l'élément qui doit être utilisé pour trier.

Ordonner une liste des tuples :

```
etudiants = [  ("Clément", 14, 16), ("Charles", 12, 15), ("Oriane", 14, 18),  ("Thomas", 11, 12) ]
sorted(etudiants)                                   # tri par ordre alphabétique sur la première colonne (nom)
sorted(etudiants, key=lambda colonnes: colonnes[2])  # tri par ordre numérique sur la troisième colonne (note)
```

La variable *"colonnes"* contiendra un élément de la liste des étudiants (c'est-à-dire un tuple).
Si on retourne *"colonnes[2]"*, cela signifie qu'on veut récupérer la moyenne de l'étudiant (troisième colonne).

Ordonner une liste d'objets :

```
class Etudiant:
    """Classe représentant un étudiant :
    prénom (attribut prenom), âge (attribut age) et sa note moyenne (attribut moyenne, entre 0 et 20)."""

    def __init__(self, prenom, age, moyenne):
        self.prenom = prenom
        self.age = age
        self.moyenne = moyenne

    def __repr__(self):
        return "<Étudiant {} (âge={}, moyenne={})>".format(self.prenom, self.age, self.moyenne)

etudiants = [ Etudiant("Clément", 14, 16), Etudiant("Charles", 12, 15), Etudiant("Oriane", 14, 18), Etudiant("Thomas", 11, 12) ]
sorted(etudiants, key=lambda etudiant: etudiant.moyenne)
sorted(etudiants, key=lambda etudiant: etudiant.age, reverse=True)          # tri dans l'ordre inverse
```

```
print(*(sorted(input(), key = lambda x : (x.isdigit(), x.isdigit() and int(x) % 2 == 0, x.isupper(), x.islower(), x))), sep='')
```

Plus rapide et plus efficace

Le module **operator** propose plusieurs fonctions qui vont s'avérer utiles pour trier des listes :

❖ **itemgetter**

```
from operator import itemgetter
sorted(etudiants, key=itemgetter(2))          # etudiants = liste des tuples; tri sur le 3eme élément du tuple
```

❖ **attrgetter**

```
from operator import attrgetter
sorted(etudiants, key=attrgetter("moyenne")) # etudiants = liste d'objets; tri sur l'attribut "moyenne" de l'objet
sorted(etudiants, key=attrgetter("age", "moyenne"))          # tri selon plusieurs critères

inventaire.sort(key=attrgetter("quantite"), reverse=True)      # si vous voulez trier par prix croissant
sorted(inventaire, key=attrgetter("prix"))                     # et par quantité décroissante => 2 sort
```

Le tri en Python est « **stable** », c'est-à-dire que l'ordre de deux éléments dans la liste n'est pas modifié s'ils sont égaux. Cette propriété permet le chaînage de tri.

• TP : dictionnaire ordonné

```
def __init__(self, base={}, **donnees):
    """Constructeur de notre objet.  Il peut ne prendre aucun paramètre  (dans ce cas, le dictionnaire sera vide)
    ou construire un  dictionnaire remplis grâce :
    -  au dictionnaire 'base' passé en premier paramètre ;
    -  aux valeurs que l'on retrouve dans 'donnees'."""

    self._cles = []              # liste contenant nos clés
    self._valeurs = []           # liste contenant les valeurs correspondant à nos clés

    if type(base) not in (dict, DictionnaireOrdonne):        # on vérifie que 'base' est un dictionnaire exploitable
       raise TypeError("le type attendu est un dictionnaire (usuel ou ordonne)")

    for cle in base:                        # on récupère les données de 'base'
       self[cle] = base[cle]

    for cle in donnees:                     # on récupère les données de 'donnees'
       self[cle] = donnees[cle]

def __add__(self, autre_objet):
    """On renvoie un nouveau dictionnaire contenant les deux dictionnaires mis bout à """
    if type(autre_objet) is not type(self):
       raise TypeError("Impossible de concaténer {0} et {1}".format(type(self), type(autre_objet)))
    else:
       nouveau = DictionnaireOrdonne()

       for cle, valeur in self.items():                    # on commence par copier self dans le dictionnaire
          nouveau[cle] = valeur

       for cle, valeur in autre_objet.items():             # on copie ensuite autre_objet
          nouveau[cle] = valeur
       return nouveau

 def __iter__(self):
    """Méthode de parcours de l'objet. On renvoie l'itérateur des clés"""
    return iter(self._cles)

def items(self):
    """Renvoie un générateur contenant les couples (cle, valeur)"""
    for i, cle in enumerate(self._cles):
       valeur = self._valeurs[i]
       yield (cle, valeur)

def __getitem__(self, cle):
    """Renvoie la valeur correspondant à la clé si elle existe, lève une exception KeyError sinon"""
    if cle not in self._cles:
       raise KeyError("La clé {0} ne se trouve pas dans le dictionnaire".format(cle))
    else:
       indice = self._cles.index(cle)
       return self._valeurs[indice]              # résultat pour l'appel objet[cle]

def sort(self):
    """Méthode permettant de trier le dictionnaire en fonction de ses clés"""
    cles_triees = sorted(self._cles)             # on trie les clés

    valeurs = []                                 # on crée une liste de valeurs, encore vide
    for cle in cles_triees:                      # on parcourt la liste des clés triées
       valeur = self[cle]
       valeurs.append(valeur)

    self._cles = cles_triees                     # enfin, on met à jour notre liste de clés et de valeurs
    self._valeurs = valeurs
```

• Itérateurs et générateurs (cachés deriere une boucle *for*)

Quand Python tombe sur une ligne du type `for element in ma_liste:`, il va appeler **l'itérateur** de ma_liste.
L'itérateur c'est un objet qui va être chargé de parcourir l'objet conteneur.
Il est créé dans la méthode spéciale **__iter__** de la classe list (dans cet exemple).
À chaque tour de boucle, Python appelle la méthode spéciale **__next__** de l'itérateur, qui doit renvoyer l'élément suivant du parcours ou lever l'exception ***StopIteration*** si le parcours touche à sa fin.

```
ma_chaine = "tuto"
iterateur_de_ma_chaine = iter(ma_chaine)      # appelle la méthode spéciale __iter__ de la chaîne
next(iterateur_de_ma_chaine)                  # => 't'   # appelle la méthode spéciale __next__ de l'itérateur
# ....
next( iterateur_de_ma_chaine)                 # => 'o'
next( iterateur_de_ma_chaine)                 # => Traceback (most recent call last): ... StopIteration
```

Notre itérateur :

```
class RevStr(str):
    """Classe reprenant les méthodes et attributs des
    chaînes construites depuis 'str'.
    On définit une méthode de parcours différente :
    on parcourt la chaine de la dernière à la première.
    Les autres méthodes ne sont pas redéfinies"""

    def __iter__(self):
        """Cette méthode renvoie un itérateur
        parcourant la chaîne dans le sens """
        return ItRevStr(self)       # renvoie l'itérateur créé
```

```
class ItRevStr:
    """Un itérateur qui parcourt une chaîne de la dernière lettre à la
    première. Attributs : position courante et la chaîne à parcourir"""

    def __init__(self, chaine_a_parcourir):
        """On se positionne à la fin de la chaîne"""
        self.chaine_a_parcourir = chaine_a_parcourir
        self.position = len(chaine_a_parcourir)

    def __next__(self):
        """Cette méthode doit renvoyer l'élément suivant dans le
        parcours, ou lever l'exception 'StopIteration' """
        if self.position == 0:                  # fin du parcours
            raise StopIteration
        self.position -= 1                      # décrémente la position
        return self.chaine_a_parcourir[self.position]
```

```
# exemple d'utilisation
ma_chaine = RevStr("Bonjour")
ma_chaine                       # => 'Bonjour'
for lettre in ma_chaine:
    print(lettre)               # => r u o j n o B
```

Les **générateurs** sont avant tout un moyen plus pratique de créer et manipuler des itérateurs.
L'idée consiste à définir ***une fonction*** pour un type de parcours, avec le mot clé ***yield*** dans son corps.

Quand on demande le premier élément du parcours (grâce à ***next***), la fonction commence son exécution.
Dès qu'elle rencontre une instruction ***yield***, elle renvoie la valeur qui suit et se met en pause.
Quand on demande l'élément suivant de l'objet (grâce, une nouvelle fois, à ***next***), l'exécution reprend à l'endroit où elle s'était arrêtée et s'interrompt au yield suivant… et ainsi de suite.
À la fin de l'exécution de la fonction, l'exception StopIteration est automatiquement levée par Python.

Notre generateur :

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre borne_inf et borne_sup (borne_inf < borne_sup) """
    borne_inf += 1
    while borne_inf < borne_sup:
        yield borne_inf
        borne_inf += 1
```

```
intervalle              # <function intervalle at ...>
intervalle(5, 9)        # <generator object intervalle...>
mon_iterateur = iter(intervalle(5, 9))
```

```
next(mon_iterateur)   # => 6
next(mon_iterateur)   # => 7
next(mon_iterateur)   # => 8
next(mon_iterateur)   # => Traceback... StopIteration
```

```
# exemple d'utilisation
for nombre in intervalle(5, 10):          # Attention : on exécute la fonction
    print(nombre)
```

Generator expressions : similaires aux compréhensions des listes:
Syntaxe : **(**expression for element in liste if predicat**)**
=> un générateur sur lequel on peut itérer pour récupérer les éléments calculés

• Les décorateurs

Les **décorateurs** sont un moyen simple de modifier le comportement « par défaut » de fonctions.
C'est un exemple assez flagrant de ce qu'on appelle la **métaprogrammation** (l'écriture de programmes manipulant… d'autres programmes)

Exemple d'un cas d'utilisation : tester les performances de certaines de nos fonctions
- une possibilité, effectivement, consiste à modifier chacune des fonctions devant intégrer ce test; mais ce n'est pas très élégant, ni très pratique, ni très sûr… bref ce n'est pas la meilleure solution.
- une autre possibilité consiste à utiliser un **décorateur** : ce décorateur se chargera d'exécuter notre fonction en calculant le temps qu'elle met et pourra, par exemple, afficher une alerte si cette durée est trop élevée. Pour indiquer qu'une fonction doit intégrer ce test, il suffira d'ajouter une simple ligne avant sa définition.

```
def monDecorateur(fonction):
    """Premier exemple de décorateur"""
    print("Décorateur appelé avec: ",fonction)
    return fonction
```

```
@monDecorateur  # équivalent à "salut = moDecorateur(salut)"
def salut():
    """Fonction modifiée par notre décorateur"""
    print("Salut !")
```

Le décorateur prend en paramètre une fonction et renvoie une fonction (la fonction appelée ou une autre).
Le moment où notre décorateur est appelé, c'est **lors de la définition** de notre fonction (et non de l'appel).

```
def monDecorateur(fonction):
    """Notre décorateur : il va afficher un message avant l'appel de la fonction définie"""

    def fonctioModifiee():                                    # définie à l'intérieur de la fonction décorateur
        """Fonction que l'on va renvoyer. Il s'agit en fait d'une version un peu modifiée de notre fonction
        originellement définie. On se contente d'afficher un avertissement avant d'exécuter la fonction originellement définie"""

        print("Attention ! On appelle {0}".format(fonction))
        return fonction()

    return fonctionModifiee

@mon_decorateur
def salut():
    print("Salut !")
```

Lors de la définition de notre fonction *salut*, on appelle notre *décorateur*.
- Python lui passe en paramètre la fonction *salut*.
- cette fois, notre décorateur ne renvoie pas *salut* mais *fonctionModifiee*.
- notre fonction *salut*, que nous venons de définir, sera donc remplacée par notre fonction *fonctionModifiee*, définie dans notre *décorateur*.

Décorateur avec un paramètre :

```
@decorateur(parametre)
def fonction(...):
    ...
```

```
def fonction(...):
    ...
fonction = decorateur(parametre)(fonction)
```

=> 3 niveaux de fonctions ....

```
""" calculer le temps mis par notre fonction pour s'exécuter"""
import time
def controler_temps(nb_secs):
    """Contrôle le temps mis par une fonction pour s'exécuter.  Si temps d'exécution > nb_secs, on affiche une alerte"""

    def decorateur(fonction_a_executer):
        """Notre décorateur. C'est lui qui est appelé directement LORS DE LA DEFINITION de (fonction_a_executer)"""

        def fonction_modifiee(*parametres, **parametres_nommes):
            """Fonction renvoyée par notre décorateur : calcule le temps mis par la fonction à s'exécuter"""
            tps_avant = time.time()                                  # avant d'exécuter la fonction
            valeur_renvoyee = fonction_a_executer(*parametres, **parametres_nommes)          # exécute la fonction
            tps_apres = time.time()
            tps_execution = tps_apres - tps_avant

            if tps_execution >= nb_secs:
                print("La fonction {0} a mis {1} pour s'exécuter".format(fonction_a_executer, tps_execution))
            return valeur_renvoyee
        return fonction_modifiee
    return decorateur
```

• Les expression régulières

Utiliser le module "**re**" (regular expressions) → methodes **search**, **findall**, **match**, **split**, **sub**, **compile**

Signe	Explication	Expression	Chaînes contenant l'expression
*	0, 1 ou plus	abc*	'ab', 'abc', 'abcc', 'abccccc'
+	1 ou plus	abc+	'abc', 'abcc', 'abccc'
?	0 ou 1	abc?	'ab', 'abc'
.*?	The lazy .*? guarantees that the quantified dot only matches as many characters as needed for the rest of the pattern to succeed. Therefore, the pattern only matches one {START}…{END} item at a time.		

Vous pouvez également contrôler précisément le **nombre d'occurrences** grâce aux accolades :

- E{4} : signifie 4 fois la lettre E majuscule ;
- E{2,4} : signifie de 2 à 4 fois la lettre E majuscule ;
- E{,5} : signifie de 0 à 5 fois la lettre E majuscule ;
- E{8,} : signifie 8 fois minimum la lettre E majuscule.

Les classes de caractères :

- [abcd] : l'une des lettres parmi a, b, c et d
- [A-Z] : une lettre majuscule
- [A-Za-z0-9] : une lettre, majuscule ou minuscule, ou un chiffre
- [A-Z]{5} : 5 lettres majuscules (qui se suivent dans une chaîne)
- **.** : n'importe quel caractère
- \d = n'importe quel chiffre (any digit)
- \D = tout sauf un chiffre (any **non** digit)
- \s = n'importe quel espace (any whitespace char)
- \S = tout sauf un espace (any **non** whitespace char)
- \w = n'importe quelle lettre ou chiffre (any alphanumeric char); équivalent avec `[a-zA-Z0-9_]`
- \W = tout sauf une lettre ou un chiffre (any **non** alphanumeric char)

Exclusion : (a | b) # a **ou** b

Echapper les caractères spéciaux :

- \\n ou r'\n'
- \. (pour détecter les points), \+ (pour détecter les plus), \? (pour détecter les points d'interrogation)

```
import re
re.search(r"abc", "defabcdefabc")      # => premiere occurrence <_sre.SRE_Match object; span=(3, 6), match='abc'>
re.search(r"abc", "abacadaeaf")        # => nothing
re.search(r"abc*", "ab")               # => <_sre.SRE_Match object; span=(0, 2), match='ab'>
re.search(r"abc*", "abccc")            # => <_sre.SRE_Match object; span=(0, 5), match='abccc'>
re.search(r"chat*", "chateau")         # => <_sre.SRE_Match object; span=(0, 4), match='chat'>
re.findall(r"[a-z]{3}", "abcccabcd")   # => ['abc', 'cca', 'bcd']
if re.match(expression, chaine) is not None:          if re.match(expression, chaine):
    # si l'expression est au début de la chaîne           # plus intuitivement
```

Groups :

```
import re
m = re.match(r'(\w+)@(\w+)\.(\w+)', 'username@hackerrank.com')
m.group(0)          # the entire match => username@hackerrank.com
m.group(1)          # the first parenthesized subgroup => 'username'
m.group(2)          # the second parenthesized subgroup => 'hackerrank'
m.group(3)          # the third parenthesized subgroup => 'com'
m.group(1,2,3)      # multiple arguments give us a tuple => ('username', 'hackerrank', 'com')
m.groups()          # => ('username', 'hackerrank', 'com')

m = re.match(r'(?P<user>\w+)@(?P<website>\w+)\.(?P<extension>\w+)','myname@hackerrank.com')
m.groupdict()       # => {'website': 'hackerrank', 'user': 'myname', 'extension': 'com'}
```

Split : items = re.**split**(r"[,:;()\.]", text)

Remplacement :

Fonction "**sub**" (substitute) qui prend trois paramètres :

- l'expression à chercher
- par quoi remplacer cette expression
- la chaîne d'origine

et renvoie la chaîne modifiée.

```
re.sub(r"(ab)", r" \1 ", "abcdef")        # => ' ab cdef' (ajoute un espace avant et après le groupe "ab")
"abcdef".replace("ab", " ab ")            # => même résultat avec la fonction "replace" de la classe "str"
```

Donner des noms à nos groupes :

Nous pouvons également donner des noms à nos groupes.
Cela peut être plus clair que de compter sur des numéros **\<numeroGroupe>**.
Pour cela, il faut faire suivre la parenthèse ouvrant le groupe

- d'un point d'interrogation,
- d'un P majuscule
- et du nom du groupe entre chevrons <>

Ex: (?P<id>[0-9]{2})

Dans l'expression de remplacement, on utilisera l'expression **\g<nomGroupe>** pour symboliser le groupe.

```
texte = "nom='Task1', id=8\nnom='Task2', id=31\nnom='Task3', id=127"
print(re.sub(r"id=(?P<id>[0-9]+)", r"id[\g<id>]", texte))    #=> nom='Task1', id[8]\nnom='Task2', id[31]\nnom='Task3', id[127]
```

Des expressions compilées :

Si, dans votre programme, vous **utilisez plusieurs fois les mêmes expressions régulières**, il peut être utile de les compiler. Le module re propose en effet de conserver votre expression régulière sous la forme d'un objet que vous pouvez stocker dans votre programme.
Si vous devez chercher cette expression dans une chaîne, vous passez par des méthodes de l'expression. Cela vous fait gagner en performances si vous faites souvent appel à cette expression.

Exemple : une expression qui est appelée quand l'utilisateur saisit son mot de passe.

- vérifier que le mot de passe fait bien six caractères au minimum
  et qu'il ne contient que des lettres majuscules, minuscules et des chiffres :

```
chn_mdp = r"^[A-Za-z0-9]{6,}$"
```

- utiliser la méthode ***compile*** du module re
- stocker la valeur renvoyée (une expression régulière compilée) dans une variable

```
exp_mdp = re.compile(chn_mdp)
```

Ensuite, vous pouvez utiliser directement cette expression compilée.
Elle possède plusieurs méthodes utiles, dont ***search*** et ***sub*** que nous avons vu plus haut.
À la différence des fonctions du module re portant les mêmes noms, elles ne prennent pas en premier paramètre l'expression (celle-ci se trouve directement dans l'objet).

```
mot_de_passe = ""
while exp_mdp.search(mot_de_passe) is None:
    mot_de_passe = input("Tapez votre mot de passe : ")
```

Exercices :

- vérifier qu'une chaîne est un numéro de téléphone (différentes saisies possibles)

  0X XX XX XX XX ou 0X-XX-XX-XX-XX ou 0X.XX.XX.XX.XX ou 0XXXXXXXXX

```
expression = r"^0[0-9]([ .-]?[0-9]{2}){4}$"
while re.search(expression, chaine) is None:
   chaine = input("Saisissez un numéro de téléphone (valide) :")
```

- vérifier un numéro de téléphone (de 10 chiffres, commençant par 7, 8 ou 9)

```
res = bool(re.search(r"^[789][0-9]{9}$", text))
```

- vérifier un id valide : 10 caractères, >= 2 lettres majuscules, >= 3 chiffres, aucun caractère qui se répète

```
if bool(re.search(r"^[a-zA-Z0-9]{10}$", id) and re.search(r"([A-Z].*){2,}", id) and re.search(r"([0-9].*){3,}", id)):
      if not bool(re.search(r"([a-zA-Z0-9]).*\1", id)):
            print("Valid")
```

- vérifier un numéro de carte de crédit valide : 16 chiffres, tiret optionnel entre chaque groupe de 4 chiffres

```
if bool(re.search(r"^([0-9]{4})([-]?[0-9]{4}){3}$", id) and re.search(r"^[456]", id)):
      id = re.sub(r"-", r"", id)
      if not bool(re.search(r"([0-9])\1{3,}", id)):                    # pas plus de 3 caractères qui se répètent
            print("Valid")
```

- vérifier une adresse email valide :

```
^[a-zA-Z0-9-_]+@{1}[a-zA-Z0-9]+\.{1}[\S]{,3}$
res = bool(re.search(r"^([a-zA-Z][a-zA-Z0-9-_\.]*)@([a-zA-Z]+)\.([a-zA-Z]{1,3})$", email))    # contraintes diff
```

- vérifier un nombre flottant :

```
^(-|\+|)[0-9]*\.{1}[0-9]+$"  ou  ^(-|\+)?[0-9]*\.{1}[0-9]+$
```

- vérifier un numéro roman:

```
re.search(r"^M{,3}(CM|CD|D?C{,3})(XC|XL|L?X{,3})(IX|IV|V?I{0,3})$", text)
```

- vérifier un code postal : (must not contain more than one alternating repetitive digit pair)

```
re.search(r"^[1-9][0-9]{5}$", code) and len(re.findall(r'([0-9])(?=.\1)', code)) <= 1
```

- retourner le premier caractère alphanumérique qui se répète :

```
re.search(r"([a-zA-Z0-9])\1", ext)
if m is not None:
      print(m.group(1))
```

- remplacer "&&" avec "and" et "||" avec "ou":

```
while re.search(r" && ", text) or re.search(r" \|\| ", text):
      text = re.sub(r" && ", " and ", text)
      text = re.sub(r" \|\| ", " or ", text)
```

- trouver les indices de début et de fin de chaîne k dans S :

```
if re.search(k, S):
      for i in range(len(S)):
            m = re.match(k, S[i:])
            if m :
                  print((i + m.start(), i + m.end() -1))
```

- remplacer les symboles consécutifs entre 2 lettres avec un espace :

```
re.sub(r"(?<=[a-zA-Z])[ !@#$%&]+(?=[a-zA-Z])", r" ", text)
```

(?<=...) Matches if the current position in the string is preceded by a match for ... that ends at the current position.
This is called a positive **lookbehind assertion**.
For example, "(?<=abc)def" will match in abcdef
(the lookbehind will back up 3 chars and check if the contained pattern matches)

(?=...) Matches if ... matches next, but doesn't consume any of the string.
This is called a **lookahead assertion**.
For example, "Isaac (?=Asimov)" will match 'Isaac ' only if it's followed by 'Asimov'.

- trouver toutes les occurrences de plus de 2 voyelles entre 2 consonnes (+ overlapping)

```
re.findall(r"(?<=[qwrtypsdfghjklzxcvbnm])[aeiou]{2,}(?=[qwrtypsdfghjklzxcvbnm])", text)

text = input().strip()
v = "[aeiouAEIOU]"
c = "[qwrtypsdfghjklzxcvbnmQWRTYPSDFGHJKLZXCVBNM]"
m = re.findall(r"(?<="+c+")"+v+"{2,}(?="+c+")", text)
```

- verifier si une chaine contient que des 'a' et 'A' et maximum 3 'a' consecutifs et maximum 3 'A' consecutifs :

```
re.search(r'^([aA](?!([^aA]|[a]{4,}|[A]{4,})))*$', text)
```

- use **inverse match** ^ (. **(?!** (some text) **)**)* $

- faut pas:
  - trouver un autre caractere que a ou A ([^aA] = n'importe quel autre caractere sauf a ou A)
  - trouver plus de 4 caracteres 'a' consecutifs
  - trouver plus de 4 caracteres 'A' consecutifs
- le **.** est remplacé par [aA] (pour s'assurer que le string contient au moins une lettre qui soit 'a' ou 'A')

- verifier s'il y a au moins 3 chiffres consecutifs egaux dans les colones d'une matrice 5x5

```
re.search(r'(\d).{4}\1.{4}\1)', text)
```

- verifier s'il y a au moins 3 chiffres consecutifs egaux dans les lignes d'une matrice 5x5

```
re.search(r"^(?:.{5})*  (  (\d)\3\3.{2} |  .{2}(\d)\4\4 |  .{1}(\d)\5\5.{1}   )", text)
```

- verifier s'il y a au moins 3 chiffres consecutifs egaux dans les lignes ou les colones d'une matrice 5x5

```
re.search(r" (^(?:.{5})*  (   (\d)\3\3.{2}  |  .{2}(\d)\4\4  |  .{1}(\d)\5\5.{1}  ) ) | (\d).{4}\6.{4}\6", text)
```

- verifier si un nombre binaire $S=a_0b_0a_1b_1...a_{n-2}b_{n-2}a_{n-1}b_{n-1}$ est un string superieur
  $A=a_{n-1}\times2^{n-1} + a_{n-2}\times2^{n-2} + ... + a_1\times2 + a_0$ >= $B=b_{n-1}\times2^{n-1} + b_{n-2}\times2^{n-2} + ... + b_1\times2 + b_0$

```
re.search(r"(10$) | (  (10 ( 00 | 11)*)(?:11$))", text)
```

- verifier si un string S qui contient que les lettres {a, b, c, d, e} verifie les 4 conditions:
  - S a le meme nombre d'occurrences de 'a' et de 'b'
  - S a le meme nombre d'occurrences de 'c' et de 'd'
  - dans chaque prefix de S, le nombre d'occurrences de a et de b differe de 1
  - dans chaque prefix de S, le nombre d'occurrences de c et de d differe de 1

```
re.search(r"^(?=([ab]*c[ab]*d|[ab]*d[ab]*c)*[ab]*$)(?=([cd]*a[cd]*b|[cd]*b[cd]*a)*[cd]*$)", text)
```

- validez le mot de passe de Scrooge qui doit respecter les contraintes :
  - avoir 20 caractères
  - avoir n'importe quel caractere sauf le retour à la ligne
  - avoir au moins 1 lettre minuscule (a-z)
  - avoir au moins 2 lettres majuscules (A-Z)
  - avoir maximum un '7'
  - avoir maximum deux '8'
  - avoir maximum trois '9'
  - des 0 positionnés sur des indices paires (2, 4, ..., 20)
  - des 6 positionnés sur des indices impaires (1, 3, ..., 19)
  - des 1 positionnés sur des indices primes (2, 3, 7, 11, 13, 17, 19)
  - '2' ne doit pas etre positionné au début ou à la fin de la chaine
  - '3, '4' et '5' ne doivent pas se suivre (i.e., 345, 354, 435, 453, 534, 543)

```
Regex_Pattern = r'^(?=[^\n]{20}$)' \
                r'(?=.*[a-z])(?=.*[A-Z].*[A-Z])' \
                 r'(?=[^12].*[^2]$)' \
                r'(?!(..)*0)' \
                 r'^(?!((..){2,}|(...){3,}|(.....){5,})(?<=1))'  \
                 r'(?!.*345)(?!.*354)(?!.*435)(?!.*453)(?!.*534)(?!.*543)' \
                 r'(?!(..)*.6)' \
                r'(?!.*7.*7)' \
                r'(?!.*8.*8.*8)' \
                r'(?!.*9.*9.*9.*9)'
```

ou

```
Regex_Pattern = r'^(?=.{20}$)
                 (?=.*?[a-z].*)
                (?=.*?[A-Z].*?[A-Z].*)
                (?!.*?(345|354|435|453|534|543))
                (?!.*?7.*?7)(?!.*?8.*?8.*?8)(?! .*?9.*?9.*?9.*?9)
          [^012][^6][^0][^61][^0][^61][^0][^61][^01][^61][^0][^61][^0][^61][^01][^61][^0][^61][^0][^612]$'
```

• Mathématique

Le module math

Fonctions usuelles :

```
import math
math.pow(5, 2)          # 5 au carré => 25.0
 5 ** 2                 # pratiquement identique à pow(5, 2) => 25 (retourne un entier, si possible)
math.sqrt(25)           # racine carrée de 25 (square root) => 5.0
math.exp(5)             # exponentielle; 148.4131591025766
math.fabs(-3)           # valeur absolue => 3.0
```

Trigonométrie :

```
# 1 rad = 57,29 degrés
math.degrees(angle_en_radians)          # convertit en degrés
math.radians(angle_en_degrés)           # convertit en radians
math.cos(...)                           # cosinus
math.sin(...)                           # sinus
math.tan(...)                           # tangente
math.acos(...)                          # arc cosinus
math.asin(...)                          # arc sinus
math.ata(...)                           # arc tangente
```

Arrondir un nombre :

```
math.ceil(2.3)          #renvoie le plus petit entier >= 2.3    => 3
math.floor(5.8)         # renvoie le plus grand entier <= 5.8  => 5
math.trunc(9.5)         # tronque 9.5 => 9
```

Constantes :

```
math.pi
math.e
```

Le module fractions

```
from fractions import Fraction
un_demi = Fraction(1, 2)                    # constructeur  => Fraction(1, 2)
un_quart = Fraction('1/4')                  # constructeur  => Fraction(1, 4)
autre_fraction = Fraction(-5, 30)           # constructeur  => Fraction(-1, 6)
Fraction.from_float(0.5)                    # depuis un flottant => Fraction(1, 2)
float(un_quart)                             # => 0.25

un_dixieme = Fraction(1, 10)
un_dixieme + un_dixieme + un_dixieme        # => Fraction(3, 10)
0.1 + 0.1 + 0.1                             # => 0.30000000000000004; <=> différence de précision

un_dixieme * un_quart                       # => Fraction(1, 40)
un_dixieme + 5                              # => Fraction(51, 10)
un_demi / un_quart                          # => Fraction(2, 1)
un_quart / un_demi                          # => Fraction(1, 2)
```

Les module random (générer des nombres pseudo-aléatoires)

```
import random
random.random()                             # génère un nombre pseudo-aléatoire compris entre 0 et 1
random.randrange(5, 10, 2)                  # générer un nombre aléatoire entre [5, 10), avec un écart de 2 entre valeurs
                                            # => chercher dans la liste des valeurs [5, 7, 9]; 3eme paramètre optionnel
random.randint(1, 6)                        # générer un nombre aléatoire entier entre 1 inclus et 6 inclus
random.choice(['a', 'b', 'k', 'p', 'i', 'w', 'z'])  # renvoie au hasard un élément d'une séquence passée en paramètre  => 'k'

liste = ['a', 'b', 'k', 'p', 'i', 'w', 'z']
random.shuffle(liste)           # => mélange la séquence passée en paramètre => ['p', 'k', 'w', 'z', 'i', 'b', 'a']; modifie 'liste'
random.sample(liste,5)          # => renvoie une nouvelle séquence de 5 éléments choisis au hasard dans la liste de ref
```

• Programmation système

Les flux standard

Trois flux standard :
- **entrée standard** (input) (par default : votre clavier)
- **sortie standard** (print) (par défaut : redirige vers l'écran)
- **erreur standard** (traceback d'une exception) (par défaut : redirige vers l'écran)

On peut accéder aux objets représentant ces flux standard grâce au module "**sys**"

```
import sys
sys.stdin          # l'entrée standard (standard input)   <_io.TextIOWrapper name='<stdin>' encoding='cp850'>
sys.stdout         # la sortie standard (standard output) <_io.TextIOWrapper name='<stdout>' encoding='cp850'>
sys.stderr         # l'erreur standard (standard error)   <_io.TextIOWrapper name='<stderr>' encoding='cp850'>

sys.stdout.write("un test")      # un test7 (7=le nombre de caractères affichés)
```

Diriger la sortie vers un fichier :

```
fichier = open('sortie.txt', 'w')
sys.stdout = fichier
print("Quelque chose...")        # écrit dans le fichier 'sortie.txt' dans le dossier courant (os.getcwd())
sys.stdout = sys.__stdout__      # rétablit la sortie standard
```

Les signaux

Les signaux sont un des moyens dont dispose le système pour communiquer avec votre programme.
Typiquement, si le système doit arrêter votre programme, il va lui envoyer un signal.

```
import signal
signal.SIGINT                    # affiche "2";  le signal SIGINT est envoyé à l'arrêt du programme
```

Intercepter le signal 'SIGINT' :

```
import sys
def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)                  # retour = 0 : tout c'est bien passé; retour ≠ 0 : il y a eu des erreurs

signal.signal(signal.SIGINT, fermer_programme)    # on connecte la fonction 'fermer_programme' au signal SIGINT
```

Interpréter les arguments de la **ligne de commande**

Python nous offre plusieurs moyens pour interpréter les arguments de la ligne de commande.
Ces arguments peuvent être des paramètres que vous passez au lancement de votre programme et qui influeront sur son exécution.

Contenus du programme "test_console.py" :

```
import sys
print(sys.argv)          # argv = argument values
```

Appel du programme avec des arguments :

```
test_console.py argument1 argument2 argument3    # => ['test_console.py', 'argument1', 'argument2', 'argument3']
test_console.py "un argument avec des espaces"   # => ['test_console.py', 'un argument avec des espaces']
```

Parfois, votre programme devra déclencher plusieurs actions en fonction du premier paramètre fourni.
Par exemple, en premier argument, vous pourriez préciser l'une des valeurs suivantes :
- *start* pour démarrer une opération,
- *stop* pour l'arrêter,
- *restart* pour la redémarrer,
- *status* pour connaître son état

Exemple avec des actions simples :

```
import sys

if len(sys.argv) < 2:
   print("Précisez une action en paramètre")
   sys.exit(1)

action = sys.argv[1]

if action == "start":
   print("On démarre l'opération")
elif action == "stop":
   print("On arrête l'opération")
elif action == "restart":
   print("On redémarre l'opération")
elif action == "status":
   print("On affiche l'état (démarré ou arrêté ?) de l'opération")
else:
   print("Je ne connais pas cette action")
```

Exemple avec des actions complexes : (module **argparse**)

Contenu du programme "test.py" :

```
import argparse

parser = argparse.ArgumentParser(description="mettre le nombre X au carré")
parser.add_argument("x", type=int, help="le nombre à mettre au carré")                  # avec valeur, sans action
parser.add_argument("-v", "--verbose", action="store_true",  help="augmente la verbosité")   # sans valeur, avec action

args = parser.parse_args()                                                              # retourne les arguments interprétés
x  = args.x
retour = x ** 2
if args.verbose:
   print("{} ^ 2 = {}".format(x, retour))
else:
   print(retour)
```

Exécution du programme :

```
>>>python code.py --help
usage: code.py [-h] x
positional arguments:
 x              le nombre à mettre au carré
optional arguments:
 -h, --help     show this help message and exit
 -v, --verbose  augmente la verbosiré
```

```
>>>python code.py 5
25
>>>python code.py 5 -v
5 ^ 2 = 25
>>>python code.py 5 --verbose
5 ^ 2 = 25
```

Nous avons ajouté une option : -v ou --verbose.
Le nom commençant par un tiret, argparse suppose qu'il s'agit d'une option facultative (cela peut être modifié).

Notez que l'on appelle la méthode add_argument avec l'argument ***action***.
L'action précisée, "store_true", permet de convertir l'option précisée en booléen :
- si l'option "-v" est précisée, alors args.verbose vaudra True ;
- si l'option "-v" n'est pas précisée, alors args.verbose vaudra False.

Autres options :

- l'argument "-v" avec une valeur

```
parser.add_argument("-v", "--verbosity", type=int, help="increase output verbosity")
```

- l'argument "-v" avec une valeur dans une liste prédéfinie

```
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2], help="increase output verbosity")
```

- l'argument "-v" qui affiche differentes informations en fonction du nombre d'appels "-v" [Ex: -v, -vv, -vvv]

```
parser.add_argument("-v", "--verbosity", action="count", default=0, help="increase output verbosity"))
if args.verbosity >= 2:
   print("Running '{}'".format(__file__))
if args.verbosity >= 1:
   print("{}^2 == ".format(args.x), end="")
```

• Le réseau

Brève présentation du réseau

On va s'attacher ici à comprendre comment faire communiquer deux applications, qui peuvent être sur la même machine mais aussi sur des machines distantes (qui se connectent grâce au réseau local ou à Internet).

Il existe plusieurs protocoles de communication en réseau.
Pour que les échanges se passent correctement, les deux parties en présence doivent parler la même langue.

Le protocole TCP (Transmission Control Protocol) : (un peu plus lent que le protocole UDP mais plus sûr)

- permet de connecter deux applications et de leur faire échanger des informations;
- ce protocole dit « orienté connexion » ;
- les applications sont connectées pour communiquer et l'on peut être sûr, quand on envoie une information au travers du réseau, qu'elle a bien été réceptionnée par l'autre application;
- si la connexion est rompue, les applications doivent rétablir la connexion pour communiquer de nouveau;

Le protocole UDP (User Datagram Protocol) :

- envoie des informations au travers du réseau sans savoir si elles sont bien réceptionnées par la cible ;
- ce protocole n'est pas connecté ;
- une application envoie quelque chose au travers du réseau en spécifiant une cible
- ce type de protocole est utile si vous avez besoin de faire transiter beaucoup d'informations au travers du réseau mais qu'une petite perte occasionnelle d'informations n'est pas très handicapante.

Clients et serveur :
Le serveur est une machine qui va traiter les requêtes de plusieurs clients

Le serveur :

- attend une connexion de la part du client
- accepte la connexion quand le client se connecte
- échange des informations avec le client
- ferme la connexion

Le client :

- se connecte au serveur
- échange des informations avec le serveur
- ferme la connexion

Etablir une connexion entre un client et un serveur en se basant sur :

- le nom d'hôte (**host name**) : identifie une machine sur internet ou sur un réseau local; représente une adresse IP de façon plus claire
- le numéro de **port** (compris entre 0 et 65535) : propre au type d'information que l'on va échanger (exemples : port 80 pour http, port 443 pour https)

Les sockets (module socket)

= objets qui permettent d'ouvrir une connexion avec une machine locale ou distante et d'échanger avec elle

Création du serveur

```
import socket

# Construire un socket
socketServeur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# socket.AF_INET = la famille d'adresses
# socket.SOCK_STREAM = le type du socket pour le protocole TCP

# Connecter un socket
socketServeur.bind(('', 12800))          # le serveur est prêt à écouter sur le port 12800 (> 1024)

# Faire écouter un socket
socketServeur.listen(5)                  # 5 = le nombre maximal qu'il peut recevoir sur ce port sans les accepter

# Accepter une connexion venant du client
clientConnecte, infosConnexion = socketServeur.accept()     # cette méthode bloque le programme;
                                                              elle attend qu'un client se connecte
print(infosConnexion)                    # affiche ('127.0.0.1', 51100) quand le client se connecte sur le serveur
```

Création du client et connexion au serveur

```
import socket

# Construire un socket
socketClient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socketClient.connect(('localhost', 12800))          # localhost = sur la même machine
```

Faire communiquer les sockets (méthodes **send** et **recv**)

```
# du coté serveur
clientConnecte.send(b"Je viens d'accepter la connexion")    # information transmise sous forme des bytes

# du coté client
msg_recu = socketClient.recv(1024)                          # 1024 = le nombre maximum de caractères à lire
print(msg_recu)                                             # => b"Je viens d'accepter la connexion
```

Fermer la connexion

```
# du coté serveur
clientConnecte.close()
# du coté client
socketClient.close()
```

Un meilleur exemple de serveur:

```
import socket
import select          # select permet d'interroger plusieurs clients dans l'attente d'un message à réceptionner,
                         sans paralyser notre programme
host = ''
port = 12800

socketServeur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socketServeur.bind((host, port))
socketServeur.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port))

serveur_lance = True
clients_connectes = []
while serveur_lance:
    # on va vérifier que de nouveaux clients ne demandent pas à se connecter (on écoute le socketServeur en lecture)
    # on attend maximum 50ms
    connexions_demandees, wlist, xlist = select.select([socketServeur], [], [], 0.05)

    for connexion in connexions_demandees:
        connexion_avec_client, infos_connexion = connexion.accept()
        clients_connectes.append(connexion_avec_client)      # on ajoute le socket connecté à la liste des clients

    # maintenant, on écoute la liste des clients connectés; les clients renvoyés par select sont ceux devant être lus (recv)
    # on attend là encore 50ms maximum;
    # on enferme l'appel à select.select dans un bloc try (si la liste de clients connectés est vide, une exception peut être levée)
    clients_a_lire = []
    try:
        clients_a_lire, wlist, xlist = select.select(clients_connectes,[], [], 0.05)
    except select.error:
        pass
    else:                                                    # on parcourt la liste des clients à lire
        for client in clients_a_lire:                        # 'client' est de type socket
            msg_recu = client.recv(1024)
            msg_recu = msg_recu.decode()                     # peut planter si le message contient des caractères spéciaux
            print("Reçu {}".format(msg_recu))
            client.send(b"5 / 5")
            if msg_recu == "fin":
                serveur_lance = False
print("Fermeture des connexions")
for client in clients_connectes:
    client.close()

connexion_principale.close()
```

Pour schématiser, **select** va écouter sur une liste de clients et retourner au bout d'un temps précisé.
Les arguments : *rlist* = en attente d'être lus, *wlist* = en attente d'être écrits, *xlist* = en attente d'une erreur *timeout*.
Select renvoie la **liste des clients qui ont un message à réceptionner**.
Il suffit de parcourir ces clients et de lire les messages en attente (grâce à recv).

• Les tests (avec unittest)

Les tests vérifient que votre code réagit comme il le devrait et qu'il continue à réagir comme il le devrait après de nouvelles améliorations.

Pour chaque fonctionnalité de votre programme, il y aura un test et le test va s'assurer que votre programme reste valide même quand vous le modifierez. Ce qui deviendra de plus en plus important au fur et à mesure que votre programme gagnera en fonctionnalités, bien entendu. Faut donc tester un code dès le début, dès les premières lignes de code.

Il existe aussi plusieurs méthodes de développement, dont le **TDD** (Test-Driven Development) qui veut que l'on écrive les tests avant d'écrire le code.

Premiers exemples de tests unitaires

Le module **unittest** de la bibliothèque standard de Python inclut le mécanisme des tests unitaires.

Voici la structure que vous rencontrerez le plus souvent :

- pour chaque fonctionnalité => un ensemble de fonctions, de classes, de modules, de packages et autre ;
- pour chaque fonctionnalité => un test qui vérifie que la fonctionnalité fait bien ce qu'on lui demande (par exemple, si une certaine fonction est appelée avec certains paramètres, elle retourne telle valeur)

Test de la fonction **random.choice**

```
import random
import unittest

class RandomTest(unittest.TestCase):
    """Test case utilisé pour tester les fonctions du module 'random'."""

    def test_choice(self):
        """Test le fonctionnement de la fonction 'random.choice'."""
        liste = list(range(10))                 # on crée une liste de 0 à 9
        item = random.choice(liste)
        self.assertIn(item, liste)              # vérifie que 'item' est dans 'liste'
```

Lancer le test :

```
unittest.main()
# => .
# => ----------------------------------------------------------------------
# => Ran 1 test in 0.003s
#=> OK
```

Le retour affiché se décompose en trois parties :
- la première ligne contient **un caractère** par test exécuté; les principaux caractères sont
  - un point (".") si le test s'est validé
  - la lettre **F** si le test n'a pas obtenu le bon résultat
  - la lettre **E** si le test a rencontré une erreur (si une exception a été levée pendant l'exécution) ;
- une ligne récapitulative du **nombre de tests exécutés** ;
- la dernière ligne récapitule le **nombre de réussites ou échecs ou erreurs**;
  si tout va bien, cette dernière ligne devrait être simplement "OK".

Les **méthodes d'assertion** du module unittest:
assertIn(), assertNotIn(), assertIs(), assertIsInstance(), assertIsNone(), assertIsNot(), assertIsNotNone(), assertEqual(), assertAlmostEqual(), assertCountEqual(), assertListEqual(), assertDictContainsSubset(), assertDictEqual(), assertFalse(), assertTrue(), assertGreater(), assertGreaterEqual(), assertRaises(), ...

Ecrire un test avec un **context manager** (**with**) :

```
    def test_sample(self):
        """Test le fonctionnement de la fonction 'random.sample'."""
        liste = list(range(10))
        extrait = random.sample(liste, 5)
        for element in extrait:
            self.assertIn(element, liste)
        with self.assertRaises(ValueError):             # si le bloc dans le context manager lève bien l'exception
            random.sample(liste, 20)                    #  => alors le test passe; sinon il ne passe pas
```

Initialiser la variable à tester dans la méthode dédiée "**setUp**" :

```
def setUp(self):                    # méthode appelée avant chaque méthode de test
    """Initialisation des tests."""
    self.liste = list(range(10))
```

Découverte automatique des tests

Lancer les tests avec unittest.main() peut s'avérer pratique, mais généralement on fera appel à la découverte automatique des tests. Cette fonctionnalité permet de rechercher tous les tests unitaires contenus dans un package et de les exécuter.

Sauvegarder le code précédent (sans la commande *unittest.main()*) dans un fichier "test_random.py".

Ouvrez la console, positionnez vous dans le répertoire courant et exécuter Python avec l'option **-m unittest** :
Sous Windows : c:\python34\python.exe -m unittest
Sous Linux : python3.4 -m unittest

L'option "-m" permet d'exécuter un module spécifique (ici "unittest").
Quand appelé directement depuis Python, unittest cherche les tests unitaires dans le dossier courant.

Vous pouvez aussi lui donner un chemin de test à exécuter, par exemple
test_random.RandomTest.test_shuffle :
- test_random est le nom du module (le nom du fichier sans l'extension) ;
- RandomTest est le nom de la classe dans notre module ;
- test_shuffle est le nom de notre méthode à exécuter.

Vos tests unitaires doivent être indépendants, c'est-à-dire qu'on peut les exécuter tout seul ou en groupe.
Ils ne doivent pas dépendre d'autres tests pour s'exécuter.

La commande *python -m unittest* explore récursivement les packages et modules à la recherche de tests.
Tous les packages sont explorés, mais les modules (comme les méthodes de test) doivent commencer par *test*.

Généralement, vous trouverez une certaine fonctionnalité (disons dans cherrypy/fonctionnalite.py) et le test de cette fonctionnalité dans un module spécifique (cherrypy/test/test_fonctionnalite.py).
Le découpage du dossier test sera souvent le même que le découpage de vos sources
(c'est plus une convention qu'une obligation).

• La programmation parallèle avec threading

Création de threads

Pour créer un thread, il faut créer une **classe** qui hérite de **threading.Thread**.
On peut redéfinir son constructeur et la méthode run (méthode appelée au lancement du thread, qui contient le code qui doit s'exécuter en parallèle du reste du programme.

Exemple :

```
import random
import sys
import time
from threading import Thread

class Afficheur(Thread):
    """Thread chargé simplement d'afficher une lettre dans la console."""

    def __init__(self, lettre):
        Thread.__init__(self)
        self.lettre = lettre

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 20:
            sys.stdout.write(self.lettre)
            sys.stdout.flush()
            attente = 0.2
            attente += random.randint(1, 60) / 100
            time.sleep(attente)
            i += 1
```

Au-dessus se trouve la **définition d'un thread** :
- le constructeur prend en paramètre la lettre à afficher; il appelle le constructeur parent Thread.__init__(self)
- la méthode run est également redéfinie

Vous avez défini le thread, mais il nous reste à le créer.
Ou plutôt, à les créer, car nous allons essayer de faire deux threads s'exécutant en même temps :

```
# Création des threads
thread_1 = Afficheur("1")
thread_2 = Afficheur("2")

# Lancement des threads
thread_1.start()
thread_2.start()

# Attend que les threads se terminent
thread_1.join()
thread_2.join()
```

La méthode ".start()" va créer un thread (une partie du code qui va pouvoir s'exécuter en parallèle) et exécuter la méthode run.
La méthode ".join()" bloque le programme et ne retourne que quand le thread est terminé. Si le programme se termine pendant que des threads tournent, les threads risquent d'être fermés brusquement.

La synchronisation des threads

Programmer plusieurs flux d'instructions apporte son lot de difficultés.
Au premier abord, cela semble très pratique d'avoir plusieurs parties de notre code qui s'exécutent en même temps. Mais le développement peut être plus compliqué en proportion.

Les locks à la rescousse
Il existe plusieurs moyens de « synchroniser » nos threads, c'est-à-dire de faire en sorte qu'une partie du code ne s'exécute que si personne n'utilise la ressource partagée.
Le mécanisme de synchronisation le plus simple est le **lock** (verrou en anglais).

C'est un objet proposé par threading qui est extrêmement simple à utiliser : au début de nos instructions qui utilisent notre ressource partagée, on dit au lock de bloquer pour les autres threads. Si un autre thread veut faire appel à cette ressource, il doit patienter jusqu'à ce qu'elle soit libérée.

```
import random
import sys
import time
from threading import Thread, RLock

verrou = RLock()                        # crée une variable de type lock

class Afficheur(Thread):
    """Thread chargé simplement d'afficher un mot dans la console."""

    def __init__(self, mot):
        Thread.__init__(self)
        self.mot = mot

    def run(self):
        """Code à exécuter pendant l'exécution du thread."""
        i = 0
        while i < 5:
            with verrou:                # on verrouille une partie de notre thread (avec le context manager, with)
                for lettre in self.mot:
                    sys.stdout.write(lettre)
                    sys.stdout.flush()
                    attente = 0.2
                    attente += random.randint(1, 60) / 100
                    time.sleep(attente)
            i += 1

# Création des threads                  # Lancement des threads               # Attend que les threads se terminent
thread_1 = Afficheur("canard")          thread_1.start()                      thread_1.join()
thread_2 = Afficheur("TORTUE")          thread_2.start()                      thread_2.join()
```

• Interfaces graphiqes avec Tkinter

Tkinter (Tk interface)

Ce module permet de créer des interfaces graphiques en offrant une passerelle entre Python et la bibliothèque Tk (même s'il n'est pas maintenu directement par les développeurs de Python).

Il est disponible sur Windows et la plupart des systèmes Unix; les interfaces que vous pourrez développer auront donc toutes les chances d'être portables d'un système à l'autre.

Notez qu'il existe d'autres bibliothèques pour créer des interfaces graphiques.
Tkinter a l'avantage d'être disponible par défaut, sans nécessiter une installation supplémentaire.

Première interface graphique

```
"""Premier exemple avec Tkinter. On crée une fenêtre simple qui souhaite la bienvenue à l'utilisateur."""
from tkinter import *

varWindow = Tk()                                        # on crée une fenêtre, racine de notre interface
varLabel = Label(varWindow, text="Salut les Zér0s !")   # on crée un label (ligne de texte) souhaitant la bienvenue
                                                        # note : le premier paramètre est l'interface racine
varLabel.pack()                                         # on affiche le label dans la fenêtre
varWindow.mainloop()                    # on démarre la boucle Tkinter qui s'interrompt quand on ferme la fenêtre
```

Etapes :
- on commence par **importer** Tkinter
- on crée un objet de la classe **Tk** (la plupart du temps, cet objet sera la fenêtre principale de notre interface)
- on crée un **Label**, c'est-à-dire un objet graphique affichant du texte
- on appelle la méthode **pack** de notre Label; cette méthode permet de positionner l'objet dans notre fenêtre
- on appelle la méthode **mainloop** de notre fenêtre racine (retourne lorsqu'on ferme la fenêtre)

Quelques précisions :
- nos objets graphiques (boutons, champs de texte, cases à cocher,…) sont appelés des **widgets**
- on peut préciser plusieurs options lors de la construction de nos widgets (ici on définit juste l'option text).

Les widgets

Pour qu'un widget apparaisse, il faut :
- qu'il prenne, en premier paramètre du constructeur, la fenêtre principale ;
- qu'il fasse appel à la méthode pack, qui permet de positionner un objet dans une fenêtre ou dans un cadre

Les labels

→ on s'en sert pour afficher du texte dans notre fenêtre, du texte qui ne sera pas modifié par l'utilisateur

```
varLabel = Label(varWindow, text="contenu de notre champ label")
varLabel.pack()                          # pack = positionne et affiche
```

Les boutons

→ des widgets sur lesquels on peut cliquer et qui peuvent déclencher des actions ou commandes

```
quitButton = Button(varWindow, text="Quitter", command= varWindow.quit)
quitButton.pack()
```

Le dernier paramètre passé à notre constructeur de Button indique l'action liée à un clic sur le bouton.
Ici, c'est la méthode quit de notre fenêtre racine qui est appelée.
Ainsi, quand vous cliquez sur le bouton Quitter, la fenêtre se ferme.

Une ligne de saisie

→ une zone de texte dans lequel l'utilisateur peut écrire

```
varText = StringVar()                          # variable Tkinter <=> la variable qui va contenir le texte de notre Entry
varEntry = Entry(varWindow, textvariable=varText, width=30)          # format : une ligne
varEntry.pack()
```

Il est possible de lier la variable *varText* à une méthode de telle sorte que la méthode soit appelée quand la variable est modifiée (l'utilisateur écrit dans le champ Entry) ===> la méthode ***trace*** de la variable.

Le widget Entry n'est qu'une zone de saisie; il pourrait être utile de lui mettre une indication auprès du champ, pour que l'utilisateur sache quoi écrire (le widget Label est le plus approprié dans ce cas).

Notez qu'il existe également le widget ***Text*** qui représente un champ de texte à plusieurs lignes.

Les cases à cocher

→ définies dans la classe Checkbutton
→ on utilise une variable pour surveiller la sélection de la case

Pour surveiller l'état d'une case à cocher (qui peut être soit active soit inactive),
on préférera créer une variable de type IntVar plutôt que StringVar (pas une obligation).

```
varCase = IntVar()
varCheck = Checkbutton(varWindow, text="Ne plus poser cette question", variable=varCase)
varCheck.pack()
```

Vous pouvez ensuite contrôler l'état de la case à cocher en interrogeant la variable :

```
varCase.get()                    # case cochée => 1; sinon => 0
```

A l'instar d'un bouton, vous pouvez lier la case à cocher à une commande (appelée quand son état change).

Les boutons radio

→ boutons généralement présentés en groupes
→ un ensemble de cases à cocher mutuellement exclusives : quand vous cliquez sur l'un des boutons, celui-ci se sélectionne et tous les autres boutons du même groupe se désélectionnent

Pour créer un groupe de boutons, il faut simplement qu'ils soient tous associés à la même variable
(là encore, une variable Tkinter); la variable peut posséder le type que vous voulez.

Quand l'utilisateur change le bouton sélectionné, la valeur de la variable change également en fonction de l'option value associée au bouton.

```
varChoix = StringVar()

choixRouge = Radiobutton(varWindow, text="Rouge", variable=varChoix, value="rouge")
choixVert = Radiobutton(varWindow, text="Vert", variable=varChoix, value="vert")
choixBleu = Radiobutton(varWindow, text="Bleu", variable=varChoix, value="bleu")

choix_rouge.pack()
choix_vert.pack()
choix_bleu.pack()
```

Pour récupérer la valeur associée au bouton actuellement sélectionné, interrogez la variable :

```
varChoix.get()
```

Les listes déroulantes

→ permet de construire une liste dans laquelle on peut sélectionner un ou plusieurs éléments.
→ ici la liste comprend plusieurs lignes et non un groupe de boutons.

```
liste = Listbox(varWindow)
liste.insert(END, "Pierre")          # END = la position à laquelle insérer l'élément; constante définie par Tkinter
liste.insert(END, "Feuille")
liste.insert(END, "Ciseau")
liste.pack()
```

Pour accéder à la sélection, utilisez la méthode ***.curselection()*** de la liste.
Elle renvoie un tuple de chaînes de caractères, chacune étant la position de l'élément sélectionné.
Par exemple, si *liste.curselection()* renvoie ('2',), c'est le troisième élément de la liste qui est sélectionné.

Organiser ses widgets

Il existe plusieurs widgets qui peuvent contenir d'autres widgets.
L'un d'entre eux se nomme **Frame**. C'est un cadre rectangulaire dans lequel vous pouvez placer vos widgets... ainsi que d'autres objets Frame si besoin est.

Si vous voulez qu'un widget apparaisse dans un cadre, utilisez le Frame comme :

```
cadre = Frame(varWindow, width=768, height=576, borderwidth=1)
cadre.pack(fill=BOTH)

message = Label(cadre, text="Notre fenêtre")
message.pack(side="top", fill=X)                    # fill rempli le widget parent en largeur (X)
```

Notez qu'il existe aussi le widget **Labelframe**, un cadre avec un titr.
Il se construit comme un Frame mais peut prendre en argument le texte représentant le titre :

```
cadre = Labelframe(..., text="Titre du cadre")
```

Exemple : si nous voulons qu'un clic sur le bouton modifie le bouton lui-même ou un autre objet, nous devons placer nos widgets dans un corps de classe. D'ailleurs, à partir du moment où on sort du cadre d'un test, il est préférable de mettre le code dans une classe.

```
from tkinter import *

class Interface(Frame):
    """Notre fenêtre principale : tous les widgets sont stockés comme attributs de cette fenêtre."""

    def __init__(self, fenetre, **kwargs):
        Frame.__init__(self, fenetre, width=768, height=576, **kwargs)
        self.pack(fill=BOTH)
        self.nb_clic = 0

        # Création de nos widgets
        self.message = Label(self, text="Vous n'avez pas cliqué sur le bouton.")
        self.message.pack()

        self.bouton_quitter = Button(self, text="Quitter", command=self.quit)
        self.bouton_quitter.pack(side="left")

        self.bouton_cliquer = Button(self, text="Cliquez ici", fg="red", command=self.cliquer)
        self.bouton_cliquer.pack(side="right")

    def cliquer(self):
        """Il y a eu un clic sur le bouton => on change la valeur du label message."""
        self.nb_clic += 1
        self.message["text"] = "Vous avez cliqué {} fois.".format(self.nb_clic)

fenetre = Tk()
interface = Interface(fenetre)
interface.mainloop()
interface.destroy()
```

Dans l'ordre :

- on crée une classe qui contiendra toute la fenêtre, qui hérite de Frame (c'est-à-dire d'un cadre Tkinter)
- dans le constructeur de la fenêtre
    - on appelle le constructeur du cadre et on pack le cadre
    - on crée les différents widgets de la fenêtre; on les positionne et on les affiche également.
- on crée une méthode bouton_cliquer:
    - est appelée quand on clique sur le bouton_cliquer
    - ne prend aucun paramètre
    - met à jour le texte contenu dans le label self.message pour afficher le nb de clics enregistrés
- on crée la fenêtre Tk qui est l'objet parent de l'interface que l'on instancie ensuite
- on rentre dans la boucle mainloop : elle s'interrompra quand on fermera la fenêtre
- ensuite, on détruit la fenêtre grâce à la méthode destroy.

• Distribuer les programmes python (avec CX_FREEZE) (faut l'instaler)

Si vous voulez distribuer votre programme, vous risquez de vous heurter au problème suivant :
- pour lancer votre code, votre destinataire doit installer Python (en plus, la bonne version)
- et si vous commencez à utiliser des bibliothèques tierces, il doit aussi les installer !

Heureusement, il existe plusieurs moyens pour produire des fichiers exécutables que vous pouvez distribuer. **Cx_freeze** est un des outils qui permet d'atteindre cet objectif.

Une version ***standalone*** de votre programme contient, en plus de votre code, l'exécutable Python et les dépendances dont il a besoin. Sur Windows, vous vous retrouverez avec un fichier *.exe* et plusieurs fichiers compagnons, bien plus faciles à distribuer et, pour vos utilisateurs, à exécuter.

Le programme résultant ne sera pas sensiblement plus rapide ou plus lent.
Il ne s'agit pas de compilation, Python reste un langage interprété et l'interpréteur sera appelé pour lire votre code, même si celui-ci se trouvera dans une forme un peu plus compressée.

Avantages de cx_Freeze
- Portabilité : cx_Freeze est fait pour fonctionner aussi bien sur Windows que sur Linux ou Mac OS ;
- Compatibilité : cx_Freeze fonctionne sur des projets Python de la branche 2.X ou 3.X ;
- Simplicité : créer son programme standalone avec cx_Freeze est simple et rapide ;
- Souplesse : vous pouvez aisément personnaliser votre programme standalone avant de le construire.

Il existe d'autres outils similaires, dont le plus célèbre est **py2exe**, que vous pouvez télécharger sur le site officiel de py2exe. Il a toutefois l'inconvénient de ne fonctionner que sur Windows et, à l'heure où j'écris ces lignes du moins, de ne pas proposer de version compatible avec Python 3.X.

Sur Windows ou Linux, la syntaxe du script est la même :

```
cxfreeze salut.py
```

Si tout se passe bien, vous vous retrouvez avec un sous-dossier *dist* qui contient les bibliothèques dont votre programme a besoin pour s'exécuter, et votre executable (sur Windows - salut.exe; sur Linux - salut).

Une seconde méthode pour utiliser cx_Freeze.

```
# le fichier setup.py
"""Fichier d'installation de notre script salut.py."""

from cx_Freeze import setup, Executable

setup(                                          # on appelle la fonction setup
    name = "salut",
    version = "0.1",
    description = "Ce programme vous dit bonjour",
    executables = [Executable("salut.py")], )
```

Cette méthode n'est pas bien plus difficile mais elle peut se révéler plus puissante à l'usage.
Cette fois, nous avons créé un fichier *setup.py* qui se charge de créer l'exécutable de notre programme.

Tout tient dans l'appel à la fonction setup. Elle possède plusieurs arguments nommés :
- name : le nom de notre futur programme.
- version : sa version.
- description : sa description.
- executables : une liste contenant des objets de type Executable, type que vous importez de cx_Freeze.

Pour créer votre exécutable, vous lancez setup.py en lui passant en paramètre la commande build.
Sur Windows: C:\python34\python.exe setup.py build
Et sur Linux : python3.4 setup.py build

Une fois l'opération terminée, vous aurez dans votre dossier un sous-répertoire *build*.
Ce répertoire contient d'autres sous-répertoires portant différents noms en fonction de votre système.

• Modules utiles en Python

string

Offre plusieurs attributs :

- string.**letters** retourne une chaine de caractères avec les lettres de l'alphabet (minuscule,majuscule)
- string.**ascii_lowercase** retourne une chaine de caractères avec les lettres de l'alphabet (minuscule)
- string.**ascii_uppercase** retourne une chaine de caractères avec les lettres de l'alphabet (majuscule)
- string.**digits** retourne une chaine de caractères avec les chiffres [0-9]
- string.**hexdigits** retourne une chaine de caractères avec les chiffres [0-9] et lettres [A-F]
- string.**octdigits** retourne une chaine de caractères avec les chiffres [0-7]
- string.**printable** retourne une chaine de caractères avec tous les caractères valides
- string.**punctuation** retourne une chaine de caractères avec tous les marques de ponctuation
- string.**whitespace** retourne une chaine de caractères avec tous les caractères de délimitation

Offre une fonction :

- string.**capwords**(text, sep) : sépare 'text' en mots, capitalise chaque mot et retourne le nouveau text

textwrap

Offre deux fonctions :

- **wrap**(string, width) retourne une liste avec des éléments de maximum 'width' chars extraits du texte
- **fill**(string, width) retourne un string avec des éléments de maximum 'width' chars extraits du texte, séparés par un "\n"

itertools

Offre des nombreuses fonctions pour travailler avec les iterateurs

- itertools.**product**(list1, list2) retourne le produit cartesian entre 'list1' et 'list2'

  product(A, B) ↔ ((x,y) for x in A for y in B)

  Pour une liste de listes : product(*list)

- itertools.**permutations**(list, length) retourne des permutations successives de taille 'length' avec les elements de la liste

  permutations([1,2,3], 2) ↔ (1,2), (1,3), (2,1), (2,3), (3,1), (3,2)

- itertools.**combinations**(list, length) retourne des combinaisons successives de taille 'length' avec les elements de la liste

  combinations([1,2,3], 2) ↔ (1,2), (1,3), (2,3) # ignore order

- itertools.**combinations_with_replacement**(list, length) retourne des combinaisons successives de taille 'length' avec les elements de la liste; les elements peuvent se répéter

  combinations([1,2,3], 2) ↔ (1,1), (1, 2), (1,3), (2,2), (2,3), (3,3)

- itertools.**groupby**(list) regroupe les elements consecutives qui sont identiques

```
for item,occ in groupby("1222311"):                # occ = iterable sur occurrences identiques consec
    print( (len(list(occ)), int(item)), end=' ')   # item = element de la liste
```

- itertools.**chain**(list1, lis2, ...) regroupe les elements des listes

```
print( list( chain( [1,2,3], [2,1], [5,7] ) ) )    # => [1, 2, 3, 2, 1, 5, 7]
```

collections

Offre des nombreux modules pour le traitement des listes :

- collections.**Counter**(list) : compte le nb d'occurrences de chaque élément de la liste

```
Counter([1,1,2,3,4,5,3,2,3,4,2,1,2,3]) => un dictionnaire
        Counter({2: 4, 3: 4, 1: 3, 4: 2, 5: 1})
        .keys() : [1, 2, 3, 4, 5]
        .values() : [3, 4, 4, 2, 1]
```

- collections.**defaultdict** : un conteneur (similaire au 'dict')

```
ddict = defaultdict(list)        # list = la classe list; pas une variable
for i in range(1, n + 1):
   x = input()
   ddict[x].append(i)
# pour [a, a, b, a, b] => ddict ['a'] =[1, 2, 4], ddict[b]=[3, 5]
```

- collections.**ordereddic** : un dictionnaire qui se rappelle de l'ordre d'insertion d'éléments

```
ddict = OrderedDict()
for i in range(1, n + 1):
   x = input()
   ddict[x].append(i)
# pour [a, a, b, a, b] => ddict ['a'] =[1, 2, 4], ddict[b]=[3, 5]
```

- collections.**namedtuple** : un tuple avec un nom et avec des noms pour les attributs

```
point = namedtuple('point', 'x,y')
pt1 = point(1,2)
pt2 = point(3,4)
print(dot_product = ( pt1.x * pt2.x ) +( pt1.y * pt2.y ))          # => 11
```

- collections.**deque** : une liste avec l'accès de deux extremités

```
d = deque()
d.append(1)          # => [1]
d.appendleft(2)      # => [2,1]
d.extend('1')        # => [2, 1, '1']                  # l'attribut doit être itérable
d.extendleft('234') # => ['4', '3', '2', 2, 1, '1']   # l'attribut doit être itérable
d.count(1)           # => 1
d.pop()              # => '1'                          # supprime et renvoie le dernier élément
d.popleft()          # => '4'                          # supprime et renvoie le premier élément
d.remove(2)          # => ['3', '2', 1]                # supprime la première occurrence de l'entier 2
d.clear()            # => []
```

queue

Offre des nombreux modules pour le traitement des queues :

- queue.**Queue**(list) : queue de type FIFO
- queue.**LifoQueue**(list) : queue de type LIFO
- queue.**PriorityQueue**(list) : queue avec des priorités

Méthodes:

```
.empty() => bool
.put(x)    => ajoute l'element x dans la queue
.get()     => supprime et retourne le premier/dernier element dans la queue
```

numpy

Offre des nombreuses fonctions pour travailler avec les tableaux de nombres.

```
import numpy as np

tableau_de_zero = np.zeros((2, 3), dtype='i')      #matrice de 2 lignes et 3 colonnes avec des entiers (initialisés 0)
tableau_de_un = np.ones((2, 3), dtype='i')         # matrice de 2 lignes et 3 colonnes avec des entiers (initialisés 1)
tableau_i = np.identity(3)                         # un tableau identité = une matrice carrée avec éléments de la
                                                     diagonale principale égaux à 1 et le reste égaux à 0
tableau1 = np.array([[3, 2], [4, 6], [8, 7]])      # initialisation d'un tableau, ligne par ligne
tableau2 = np.array([1,2,3,4,5,6], float)          # initialisation d'un tableau de flottants
tableau3 = np.eye(8, 7, k =1)                      # renvoie une matrice 2-D, avec des 1 sur la diagonale et des 0
ailleurs; la diagonale principale peut être supérieure (k > 0) ou inférieure (k< 0) en fonction du paramètre k en option
print(tableau1.shape)                              # affiche le tuple avec la taille du tableau => (3, 2) (3 l, 2 c)
print(tableau2.shape)                              # => (5,)  (5 lignes, 0 colonnes)
tableau2.shape = (3, 2)                            # change la forme du tableau => [[1 2] [3 4] [5 6]]  (3 l, 2 c)
np.reshape(tableau2, (3, 2))                       # crée un nouveau tableau avec la forme (3, 2) et élems de tableau2
np.transpose(tableau1)                             # crée un nouveau tableau transposé => [[3 4 8] [2 6 7]]
print(tableau1.flatten())                          # crée une liste avec les éléments du tableau => [3 2 4 6 8 7]
print(np.concatenate((tableau1, tableau1), axis =1) # concaténer les tableaux; axe optionnelle; par default 1ère dim

Toutes les opérations mathématiques fonctionnent entre deux arrays a et b:
a + b, a - b, a * b, a // b, a / b, a % b, a ** b, ....

Les produits 'dot', 'cross', 'inner' et 'outer' entre deux arrays a et b:
np.dot(a, b)                                       # la matrice du produit entre a et b
np.cross(a, b)                                     # le produit des éléments de la diagonale principale minus
                                                     le produit des éléments de la diagonale secondaire
np.inner(a, b)                                     # le produit 'inner' entre a et b
np.outer(a, b)                                     # la matrice du produit 'outer' entre a et b

Arrondir les flottants d'un array a:
np.floor(a)
np.ceil(a)
np.rint(a)                                         # arrondi à l'entier le plus proche

Somme et produits des éléments d'un array a:
np.sum(a, axis = 0), np.sum(a)                     # la somme des éléments sur l'axe 0 ou de tous les éléments
np.prod(a, axis = 1), np.prod(a)                   # le produit des éléments sur l'axe 1ou de tous les éléments

Minimum et maximum des éléments d'un array a:
np.min(a, axis = 0), np.min(a)                     # le minimum des éléments sur l'axe 0 ou de tous les éléments
np.max(a, axis = 1), np.max(a)                     # le maximum des éléments sur l'axe 1 ou de tous les éléments

Moyenne, variance, écart-type des éléments d'un array a:
np.mean(a, axis = 0) , np.mean(a)                  # la moyenne des éléments sur l'axe 0 ou de tous les éléments
np.var(a, axis = 1), np.var(a)                     # la variance des éléments sur l'axe 1 ou de tous les éléments
np.std(a, axis = 1), np.std(a)                     # l'écart-type des éléments sur l'axe 1 ou de tous les éléments


Polynomiaux:
numpy.poly(a)                                      # les coefficients d'un polynomial avec la séquence des racines 'a'
numpy.roots(a)                                     # les racines d'un polynomial avec la séquence des coefficients 'a'
numpy.polyint(a)                                   # l'antidérivé d'un polynomial
numpy.polyder(a)                                   # le dérivé d'un polynomial
numpy.polyval(a, x)                                # évalue le polynôme à la valeur x
numpy.polyfit(a, b, x)                             # adapte un polynôme à un ensemble de données en utilisant une
                                                     approche des moindres carrés
Calculs d'algèbre linéaire:
numpy.linalg.det(a)                                # le déterminant d'un array
numpy.linalg.eig(a)                                # la valeur 'eigenvalue' d'un arrau
numpy.linalg.inv(a)                                # l'inverse d'un array a^-1
```

matplotlib

Offre des nombreuses fonctions pour créer des graphiques.

time

Représenter une date et une heure dans **un nombre unique** (***timestamp***) :
- représenter une date et une heure en fonction du nombre de secondes écoulées depuis une date précise
- la plupart du temps, cette date est l'Epoch Unix, le 1er janvier 1970 à 00:00:00.

```
import time
debut= time.time() # => 1459344492.9333987          fin > debut          # => True
fin= time.time()   # => 1459344503.1569836          fin - debut          # => 10.2235848903656
```

Représenter la date et l'heure de façon **plus présentable** :
- time.**localtime()** = une sortie sous la forme d'un objet contenant beaucoup d'informations
- elle renvoie un objet contenant, dans l'ordre :
  - tm_year : année (entier)
  - tm_mon : numéro du mois [1, 12]
  - tm_mday : numéro du jour du mois [1, 31]
  - tm_hour : heure du jour [0,23]
  - tm_min : nombre de minutes [0, 59]
  - tm_sec : nombre de secondes [0, 59]
  - tm_wday : jour de la semaine [0, 6]; 0 = lundi
  - tm_yday : jour de l'année [1, 366]
  - tm_isdst : entier; changement d'heure local

```
time.localtime()
# => time.struct_time(tm_year=2016, tm_mon=3, tm_mday=30,
                      tm_hour=15, tm_min=30, tm_sec=49, tm_wday=2, tm_yday=90, tm_isdst=1)
```

Récupérer un **timestamp depuis une date** et vice-versa :

```
print(debut)                          # => 1459344492.9333987
tempsDebut = time.localtime(debut)    # => tm_year=2016, tm_mon=3, tm_mday=30, tm_hour=15, tm_min=28, tm_sec=12, ...
secDebut = time.mktime(tempsDebut)    # => 1459344492.9333987
```

Mettre en **pause l'exécution du programme** pendant un temps déterminé => fonction **sleep**

```
time.sleep(3.5)                       # faire une pause pendant 3,5 secondes
```

Formater un temps (avec la fonction **strftime**) :
- elle permet de formater une date et heure en la représentant dans une chaîne de caractères
- elle prend deux paramètres : la chaîne de formatage et un temps *optionnel*

```
time.strftime('%A')   # => nom du jour de la semaine      time.strftime('%M')  # => minute  (entre 00 et 59)
time.strftime('%B')   # => nom du mois                    time.strftime('%S')  # => seconde (entre 00 et 59)
time.strftime('%d')   # => jour du mois (de 01 à 31)      time.strftime('%Y')  # => année
time.strftime('%H')   # => heure (de 00 à 23)
```

Donc pour afficher la date telle qu'on y est habitué en France :

```
time.strftime("%A %d %B %Y %H:%M:%S")      # => 'Wednesday 30 March 2016 16:31:53'
```

datetime (propose plusieurs classes pour représenter des dates et des heures)

Classe "date" : pour représenter une date (un jour)

```
import datetime
date = datetime.date(2010, 12, 25)          # => crée un objet   datetime.date(2010-12-25)
date.today()                                # => datetime.date(2016, 3, 30)
aujourdhui = datetime.date.today()          # => datetime.date(2016, 3, 30)
datetime.date.fromtimestamp(time.time())    # => datetime.date(2016, 3, 30)
```

Classe "time" : pour représenter une heure

On construit une heure avec cinq paramètres, tous optionnels :
- hour (0 par défaut) : les heures [0, 23]
- minute (0 par défaut) : les minutes [0,59]
- second (0 par défaut) : les secondes [0, 59]
- microsecond (0 par défaut) : la précision de l'heure en micro-secondes [0,1.000.000]
- tzinfo (None par défaut) : l'information de fuseau horaire.

Classe "datetime" : pour représenter une date et une heure

```
import datetime
datetime.datetime.now()   # renvoie l'objet avec la date et l'heure actuelles: datetime.datetime(2016, 3, 30, 16, 48, 28, 413828)
datetime.fromtimestamp(timestamp) #renvoie la date et l'heure d'un timestamp précis
```

Gestion des mots de passe : getpass et hashlib

Receptionner un mot de passe saisi par l'utilisateur (module getpass)

```
from getpass import getpass
mot_de_passe = getpass()                                        # getpass.getpass()
mot_de_passe = getpass.getpass("Tapez votre mot de passe : ")   # texte optionnel
```

Chiffrer un mot de passe (module hashlib)

```
import hashlib
hashlib.algorithms_guaranteed   # la liste des algorithmes disponibles sur toutes les plateformes (pour programmes portables)
                                #  {'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}
hashlib.algorithms_available    # la liste des algorithmes disponibles sur notre plateforme (ici windows)
                                # {'SHA256', 'MD5', 'SHA384', 'SHA1', 'sha256', 'md5', 'RIPEMD160', 'SHA224',
                                   'dsaWithSHA', 'ecdsa-with-SHA1', 'dsaEncryption', 'SHA512', 'sha1', 'sha384', 'SHA',
                                   'sha', 'whirlpool', 'DSA-SHA', 'ripemd160', 'sha512', 'sha224', 'MD4', 'md4', 'DSA'}

mot_de_passe = hashlib.sha1(b"mot de passe")  # b"..." => passe la chaine de caractères dans une chaine de bytes
mot_de_passe.hexdigest()                      # pour obtenir le chiffrement associé à cet objet => 'b47ea83257...'
mot_de_passe.digest()                                 # renvoie une chaine de bytes
```

Exercice : vérifier si le mot de passe saisi par l'utilisateur correspond au chiffrement conservé

```
import hashlib
from getpass import getpass

chaine_mot_de_passe = b"azerty"
mot_de_passe_chiffre = hashlib.sha1(chaine_mot_de_passe).hexdigest()

verrouille = True
while verrouille:
    entre = getpass("Tapez le mot de passe : ")     # azerty
    entre = entre.encode()                          # on encode la saisie pour avoir un type bytes

    entre_chiffre = hashlib.sha1(entre).hexdigest()
    if entre_chiffre == mot_de_passe_chiffre:
        verrouille = False
    else:
        print("Mot de passe incorrect")

print("Mot de passe accepté...")
```

• Des bonnes pratiques

Plusieurs conventions nous sont proposées au travers de **PEP** (Python Enhancement Proposal)

La PEP 20 : The Zen of Python

La PEP 20 nous donne des conseils très généraux sur le développement

- beautiful is better than ugly
- explicit is better than implicit
- simple is better than complex
- complex is better than complicated
- flat is better than nested : moins littéralement, du code trop imbriqué (par exemple une boucle imbriquée dans une boucle imbriquée dans une boucle…) est plus difficile à lire ;
- sparse is better than dense
- readability counts
- special cases aren't special enough to break the rules
- ..... although practicality beats purity : même si l'aspect pratique doit prendre le pas sur la pureté. Moins littéralement, il est difficile de faire un code à la fois fonctionnel et « pur » ;
- errors should never pass silently
- ..... unless explicitly silenced
- in the face of ambiguity, refuse the temptation to guess
- there should be one -- and preferably only one -- obvious way to do it
- ..... although that way may not be obvious at first
- now is better than never
- ..... although never is often better than *right* now
- if the implementation is hard to explain, it's a bad idea
- if the implementation is easy to explain, it may be a good idea
- namespaces are one honking great idea -- let's do more of those

La PEP 8 : des conventions précises

Elle nous donne des conseils très précis sur la forme du code pour améliorer la lisibilité du code.

Forme du code

- indentation : utilisez 4 espaces par niveau d'indentation
- tabulations ou espaces : ne mélangez jamais, dans le même projet, des indentations à base d'espaces et d'autres à base de tabulations; à choisir, on préfère généralement les espaces mais les tabulations peuvent être également utilisées pour marquer l'indentation
- longueur maximum d'une ligne : limitez vos lignes à un maximum de 79 caractères.
  De nombreux éditeurs favorisent des lignes de 79 caractères maximum. Pour les blocs de texte relativement longs (docstrings, par exemple), limitez-vous de préférence à 72 caractères par ligne.

Découpez vos lignes en utilisant des parenthèses, crochets ou accolades plutôt que l'anti-slash \.

```
appel_d_une_fonction(parametre_1, parametre_2,
    parametre_3, parametre_4):
```

Si vous devez découper une ligne trop longue, faites la césure après l'opérateur, pas avant.

```
# oui
un_long_calcul = variable + \
    taux * 100
```

```
# non
un_long_calcul = variable \
    + taux * 100
```

Sauts de ligne :

- séparez par deux sauts de ligne la définition d'une fonction et la définition d'une classe
- les définitions de méthodes au cœur d'une classe sont séparées par une ligne vide
- des sauts de ligne peuvent également être utilisés pour délimiter des portions de code

Encodage :

- à partir de Python 3.0, il est conseillé d'utiliser, dans du code comportant des accents, l'encodage Utf-8.

Directives d'**importation**

- les directives d'importation doivent préférentiellement se trouver sur plusieurs lignes;

```
# oui                          #non
import os                      import os, sys
import sys
```

cette syntaxe est cependant acceptée quand on importe certaines données d'un module :

```
from subprocess import Popen, PIPE
```

- les directives d'importation doivent toujours se trouver en tête du fichier, sous la documentation éventuelle du module mais avant la définition de variables globales ou de constantes du module
- les directives d'importation doivent être divisées en trois groupes, dans l'ordre :
  - les directives d'importation faisant référence à la bibliothèque standard ;
  - les directives d'importation faisant référence à des bibliothèques tierces ;
  - les directives d'importation faisant référence à des modules de votre projet.
- il devrait y avoir un saut de ligne entre chaque groupe de directives d'importation
- dans vos directives d'importation, utilisez des chemins absolus plutôt que relatifs

```
# oui                                  # non
from paquet.souspaquet import module   from . import module
```

Le **signe espace** dans les expressions et instructions

Évitez le signe espace dans les situations suivantes :

- au cœur des parenthèses, crochets et accolades :

```
# oui                          # non
spam(ham[1], {eggs: 2})        spam( ham[ 1 ], { eggs: 2 } )
```

- juste avant une virgule, un point-virgule ou un signe deux points :

```
# oui                          # non
if x == 4: print x, y; x, y = y, x   if x == 4 : print x , y ; x , y = y , x
```

- juste avant la parenthèse ouvrante qui introduit la liste des paramètres d'une fonction :

```
# oui                          # non
spam(1)                        spam (1)
```

- juste avant le crochet ouvrant indiquant une indexation ou sélection :

```
# oui                          # non
dict['key'] = list[index]      dict ['key'] = list [index]
```

- plus d'un espace autour de l'opérateur d'affectation = (ou autre) pour l'aligner avec une autre instruction

```
# oui                          # Non
x = 1                          x         = 1
y = 2                          y         = 2
long_variable = 3              long_variable = 3
```

- toujours entourer les opérateurs suivants d'un espace (un avant le symbole, un après) :
  affectation : =, +=, -=, etc. ;
  comparaison : <, >, <=, …, in, not in, is, is not ;
  booléens : and, or, not ;
  arithmétiques : +, -, *, etc.

```
# oui                          # non
i = i + 1                      i=i+1
submitted += 1                 submitted +=1
x = x * 2 - 1                  x = x*2 - 1
hypot2 = x * x + y * y         hypot2 = x*x + y*y
c = (a + b) * (a - b)          c = (a+b) * (a-b)
```

Attention : n'utilisez pas d'espaces autour du signe = si c'est dans le contexte d'un paramètre ayant une valeur par défaut (définition d'une fonction) ou d'un appel de paramètre (appel de fonction).

```
# oui                          # non
def fonction(parametre=5):     def fonction(parametre = 5):
    ...                            ...
fonction(parametre=32)         fonction(parametre = 32)
```

❖ Il est déconseillé de mettre plusieurs instructions sur une même ligne :

```
# oui
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

```
# non
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

Commentaires :

- les commentaires qui contredisent le code sont pires qu'une absence de commentaire.
- lorsque le code doit changer, faites passer parmi vos priorités absolues la mise à jour des commentaires !
- les commentaires doivent être des phrases complètes, commençant par une majuscule.
- le point terminant la phrase peut être absent si le commentaire est court.
- si vous écrivez en anglais, suivez les règles définies par Strunk and White dans "The Elements of Style"

Conventions de nommage

❖ n'utilisez jamais les caractères suivants de manière isolée comme noms de variables :
- l (L minuscule), O (o majuscule) et I (i majuscule)
- dans certaines polices ils peuvent être aisément confondus avec les chiffres 0 ou 1.

❖ noms de **packages** et **modules**
- les modules et packages doivent avoir des noms courts, constitués de lettres minuscules.
- les noms de modules peuvent contenir des signes _ (souligné).
- bien que les noms de packages puissent également en contenir, la PEP 8 nous le déconseille.

❖ noms de **classes**
- les noms de classes utilisent la convention suivante : la variable est écrite en minuscules, exceptée la première lettre de chaque mot qui la constitue (exemple : MaClasse)

❖ noms d'**exceptions**
- les exceptions étant des classes, elles suivent la même convention
- en anglais, si l'exception est une erreur, on fait suivre le nom du suffixe Error (vous retrouvez cette convention dans SyntaxError, IndexError…)

❖ noms de **fonctions**, **méthodes** et **variables**
- le nom est entièrement écrit en minuscules et les mots sont séparés par des signes soulignés (_) (exemple : nom_de_fonction)

❖ noms de **constantes**
- les constantes doivent être écrites entièrement en majuscules, les mots étant séparés par un signe souligné (_). (exemple : NOM_DE_MA_CONSTANTE)

Conventions de programmation

❖ comparaisons
- les comparaisons avec des singletons (comme None) doivent toujours se faire avec les opérateurs is et is not, jamais avec les opérateurs == ou !=.

```
# oui
if objet is None:
    ...
```

```
# non
if objet == None:
    ...
```

- quand cela est possible, utilisez l'instruction **if objet:** si vous voulez dire if objet is not None:.
- la vérification du type d'un objet doit se faire avec la fonction isinstance :

```
# oui
if isinstance(variable, str):
    ...
```

```
# non
if type(variable) == str:
    ...
```

- quand vous comparez des séquences, utilisez le fait qu'une séquence vide est False.

```
if liste:    # la liste n'est pas vide
```

- ne comparez pas des booléens à True ou False :

```
# oui                                   # Non
if booleen:       # si booleen est vrai     if booleen == True:
    ...                                       ...
if not booleen:   # si booleen n'est pas vrai  # Encore pire
    ...                                       if booleen is True:
                                                  ...
```

La PEP 257 : de belles documentations

Définit d'autres conventions concernant la documentation via les **docstrings**.

La *docstring* (chaîne de documentation) est une chaîne de caractères placée juste après la définition d'un module, d'une classe, fonction ou méthode; l'attribut spécial __doc__ de l'objet.

Tous les modules doivent être documentés grâce aux docstrings.
Les fonctions et classes exportées par un module doivent également être documentées ainsi.
Cela vaut aussi pour les méthodes publiques d'une classe (y compris le constructeur __init__).
Un package peut être documenté via une docstring placée dans le fichier __init__.py.

Pour des raisons de cohérence, utilisez toujours des guillemets triples """ autour de vos docstrings.

Les docstrings sur une seule ligne

```
def kos_root():
   """Return the pathname of the KOS root directory."""
   global _kos_root
   if _kos_root: return _kos_root
   …
```

Notes
- les guillemets triples sont utilisés même si la chaîne tient sur une seule ligne
  (plus simple de l'étendre par la suite)
- les trois guillemets """ fermant la chaîne sont sur la même ligne que les trois guillemets qui l'ouvrent
  (ceci est préférable pour une docstring d'une seule ligne)
- il n'y a aucun saut de ligne avant ou après la docstring.
- la chaîne de documentation est une phrase, elle se termine par un point ..
- la docstring sur une seule ligne ne doit pas décrire la signature des paramètres
  à passer à la fonction/méthode, ou son type de retour

  N'écrivez pas :

```
def fonction(a, b):
   """fonction(a, b) -> list"""
```

  Cette syntaxe est uniquement valable pour les fonctions C (comme les built-ins).
  Pour les fonctions Python, l'introspection peut être utilisée pour déterminer les paramètres attendus.
  L'introspection ne peut cependant pas être utilisée pour déterminer le type de retour de la fonction/méthode. Si vous voulez le préciser, incluez-le dans la docstring sous une forme explicite :

```
"""Fonction faisant cela et renvoyant une liste."""
```

Les docstrings sur plusieurs lignes

```
class MaClasse:
   def __init__(self, ...):
      """Constructeur de la classe MaClasse

      Une description plus longue...
      sur plusieurs lignes...

      """
```

Les docstrings sur plusieurs lignes sont constituées
- d'une première ligne résumant brièvement l'objet (fonction, méthode, classe, module),
- suivie d'un saut de ligne,
- suivi d'une description plus longue.

Autres conventions :
- la première ligne de la docstring peut se trouver juste après les guillemets ouvrant la chaîne ou juste en-dessous; le reste de la docstring doit être indenté au même niveau que la première ligne :
- insérez un saut de ligne avant et après chaque docstring documentant une classe
- la docstring d'un module doit généralement dresser la liste des classes, exceptions et fonctions, ainsi que des autres objets exportés par ce module (une ligne de description par objet); cette ligne de description donne généralement moins d'informations sur l'objet que sa propre documentation.
- la documentation d'un package (la docstring se trouvant dans le fichier __init__.py) doit également dresser la liste des modules et sous-packages qu'il exporte.
- la documentation d'une fonction ou méthode doit décrire son comportement et documenter ses arguments, sa valeur de retour, ses effets de bord, les exceptions qu'elle peut lever et les restrictions concernant son appel (quand ou dans quelles conditions appeler cette fonction); les paramètres optionnels doivent également être documentés.

```
def complexe(reel=0.0, image=0.0):
    """Forme un nombre complexe.

    Paramètres nommés :
    reel -- la partie réelle (0.0 par défaut)
    image -- la partie imaginaire (0.0 par défaut)

    """
    if image == 0.0 and reel == 0.0: return complexe_zero
    ...
```

- la documentation d'une classe doit, de même, décrire son comportement, documenter ses méthodes publiques et ses attributs.

Le BDFL nous conseille de sauter une ligne avant de fermer nos docstrings quand elles sont sur plusieurs lignes. Les trois guillemets fermant la docstring sont ainsi sur une ligne vide par ailleurs.

```
def fonction():
    """Documentation brève sur une ligne.

    Documentation plus longue...

    """
```