

Prise en main de Python

INF TC1
Septembre 2015

Version Élèves

2015-16

INF TC1

I Modalités des travaux à rendre

Modalités pratiques pour les devoirs à rendre

- Il y a deux devoirs à rendre à l'issue des séances de l'action INF-TC1. Les enseignants vous les indiquerons. Un contrôle continu "surprise" est également prévu. Il aura lieu dans une des séances de TD/BE.
- Pour les travaux à rendre, les élèves doivent se mettre par binôme dans un même groupe de TD. Il est néanmoins possible de rendre seul un devoir (cf. également si le nombre d'élèves d'un groupe est impair).

Comment rendre le travail

Le travail est rendu sous forme électronique consistant en un fichier archive (fichier "zip" / "tgz" / ...) qui sera déposé sur le site "pedagogie ", rubrique "Travaux " pour l'action de formation "INF TC1- Algorithmique et Structure de données". Ce fichier devra comporter uniquement :

- les fichiers sources commentés (.py) que vous avez écrits,
- un rapport au format "pdf " (sauf pour le premier devoir) qui doit comporter :
 - sur la première page : les noms des élèves, le numéro du groupe de TD, le nom de l'enseignant et le titre du TD.
 - un compte-rendu de votre travail détaillant l'analyse du problème, sa modélisation, les solutions apportées, etc...
- les codes sources commentés de vos programmes (.py)
- les traces d'exécution
- un commentaire des résultats obtenus

Attention de bien vérifier que la version du code présentée dans le rapport est identique à celle des fichiers sources (.py) joints à l'archive.

Délais :

Sauf indication contraire transmis par mail ou par l'enseignant de votre groupe, ces comptes-rendus devront être déposés sur le site pedagogie dans un délai de deux semaines après la séance de TD correspondant.

⚠ ATTENTION : les soumissions tardives des travaux ne sont pas acceptées.

Instructions pour préparer le fichier archive

Créer le fichier archive (par exemple fichier ".zip") :

- 1) Exécuter le logiciel de compression-archivage (*Winzip* ou *Winrar* sous windows)
- 2) Ajouter à l'archive les fichiers indiqués ci-dessus
- 3) Nommer votre archive : *eleve1-eleve2-i.zip* où :
 - i est le numéro du devoir,
 - eleve1 et eleve2 sont les noms des 2 élèves séparés par un trait-d'union (ne pas mettre d'accents et les noms trop longs peuvent être abrégés!).

Remarque pour créer une archive sous Mac OSX ou Linux : clique droit sur un dossier + compresser ou bien par la commande *tar* dans une console.

II Objectifs de ce BE

- Apprentissage par l'exemple du langage Python
- Acquisition des bases pour la suite des enseignements d'Informatique à l'ECL.

II-1 Principe de ce BE

- Il vous est proposé des exemples à étudier / exécuter. Ces exemples vous apportent les informations nécessaires pour réaliser les exercices demandés.
- Ne passer pas d'un exemple au suivant sans l'avoir compris. Le but n'est pas de sauter aux exercices difficiles mais de progresser réellement !
- Il est prévu deux types d'exercices :
 - Ceux portant la mention "Bonus" : si vous n'avez pas fait trop de Python, ne les traitez pas à la première lecture. Vous pouvez néanmoins les traiter en tant que Bonus (de note).
 - Ceux ne portant pas la mention "Bonus" : à faire !
- Les sections portant la mention "Complément" (d'information pour traiter les exercices) peuvent être lues dans une seconde lecture.

II-2 Documentation

- Il existe une pléthore de documents et sites sur Python.
- Il vous sera proposé un document d'initiation à Python qui accompagnera le support du cours INF-TC1.

II-3 A savoir

- Version de Python utilisée : 3.4
- Il est préférable d'utiliser un environnement intégré composé d'un éditeur, d'un interpréteur Python et d'un outil de débogage (IDE : Integrated Development Env.).

Dans le domaine des logiciels libres :

- L'IDE utilisé prévu est **spyder** (qui vient avec *Anaconda*). Il est quelque peu lourd mais facilite le débogage (debuggage) d'un code.
- Il y a également l'éditeur **iep** (assez léger et complet, difficulté de débogage du code), **Pyzo** ou **canopy** (problème pour déboguer) ou encore **ninja** (très léger mais avec moins de services).

III Introduction

- Python est un langage de programmation à la fois impératif, fonctionnel et objet avec un typage dynamique. Dans Python, toute expression est une valeur (que l'on peut stocker / affecter / écrire / ... ou pas).
- Penser à utiliser la fonction **help** pour connaître les détails d'une fonction : par exemple `help('print')` ou `print?` (ou encore `print??`)
- Si vous travaillez sans un IDE (éditeur avec interprète de Python intégré), préférez *IPython* à l'interprète de base de Python. IPython fournit des outils supplémentaires ainsi qu'un accès au système hôte (Windows, Linux, MacOS, etc).

III-1 Commentaires en Python

- Deux manières pour écrire un commentaire en Python :

```
# En Python, un commentaire court commence par un # et se limite à la fin de la ligne
```

```
"""
```

```
On peut aussi créer un commentaire long = une zone de  
commentaires sur plusieurs lignes placées  
entre une paire de triple-guillemets dont le second suit :  
"""
```

☞ Remarques :

- Un `' ; '` à la fin d'une ligne rend la ligne muette (pas d'affichage des résultats). Il permet également de placer plusieurs expressions sur la même ligne.
- Si vous placez un commentaire immédiatement après la signature d'une fonction (ou une méthode, une classe, etc.) entre un couple de `"""`, votre commentaire (appelé *docstring*) devient accessible avec *help* :
- Python est un langage à indentation : un bloc est une zone contiguë de lignes d'expressions avec la même indentation. Un tel bloc peut en contenir d'autres.
→ Essayer d'utiliser la même indentation partout (espaces ou tabulation). Une mauvaise indentation qui ne provoque pas d'erreur de syntaxe est un piège parfois difficile à détecter.

IV Exemples et exercices simples

IV-1 Exemple 1 : moyenne

- Calcul de la moyenne de deux entiers

```
a=input('Donner la valeur de a : ')
b=input('Donner la valeur de b : ')
print('la moyenne = ', (int(a) + int(b))/2)      # int(a) : conversion de 'a' en entier
```

- Une seconde méthode : on convertit en lisant les valeurs en int

```
a=int(input('Donner la valeur de a : '))
b=int(input('Donner la valeur de b : '))
print('la moyenne = ', (a+b)/2)
```

Points Python :

- input
- Conversion en int.
- print
- Affectation (=)

☞ Remarque : que se passe-t-il si au lieu de donner un entier, on donne une lettre ?

→ Voir plus loin les *exceptions* section [XV](#) page [27](#)

IV-2 Exemple 2 : racines carrées

- Lire les coefficients a, b, c et calculer et afficher les racines de l'équation quadratique $ax^2 + bx + c = 0$

```
import math;

a=int(input('Donner le coefficient a : '))
b=int(input('Donner le coefficient b : '))
c=int(input('Donner le coefficient c : '))

if (a == 0):
    if (b != 0):
        print("Pas quadratique : racine simple x = ", -c/b)
    else:
        print("une blague ! ")
else:
    delta=b*b-4*a*c    #ou b**2
    if(delta < 0):
        print("pas de racines reelles")
    else:
        assert(a != 0)    # Déjà vérifié mais pour montrer l'intérêt de "assert".
        if(delta > 0):
            x1 = (-b+math.sqrt(delta))/(2*a)
            x2 = (-b-math.sqrt(delta))/(2*a)
            print("x1 = ", x1)
            print("x2 = ", x2)
        else:
            x1 = x2 = -b/(2*a)
            print("racine double x1 = x2 = ", x1)
```

- Après un ':' (d'ouverture d'un block), si vous passez à la ligne, n'oubliez pas l'indentation.

Points Python :

- import
- structures conditionnelles if-else, if-elif-else, **assert**
- indentation (préférez des espaces à la tabulation) : espaces ou tabulation, utilisez la même chose dans un bloc !
- test et comparaisons (==, !=, <, >, >=, <=, ...)
- math.sqrt()

☞ Remarques :

- Contrairement aux versions antérieures à Python 3, le résultat de l'opérateur '/' est un réel. Pour récupérer la partie entière d'une division, utiliser '//' (ou convertir le quotient en un entier).

- La notation **math.sqrt(delta)** : pour éviter de mentionner **math**, remplacer

```
import math;           par
from math import sqrt;
```

(voir aussi plus loin à propos des modules)

IV-3 Exercice : pair/impair

- Lire un entier au clavier et décider (et afficher) s'il est pair ou impair.

V Introduction aux fonctions

- Une fonction est définie par `def nom_fonction(paramètres formels) :`
`| -indent-| Corps de la fonction`
 Cette fonction sera appelée par `nom_fonction(arguments d'appel)`

V-1 Exemple : moyenne de deux entiers

- Calcul de la moyenne de deux entiers par une fonction (de signature $\text{entier} \times \text{entier} \rightarrow \text{reel}$)

```
def moyenne(a,b):
    return (a+b)/2 # indentation obligatoire si vous passez à la ligne après ':'

# — Partie Principale —
a=input('Donner la valeur de a : ')
b=input('Donner la valeur de b : ')
print('la moyenne = ', moyenne(int(a),int(b)))
```

→ Et pour la signature $\text{moyenne} : \text{entier} \times \text{entier} \rightarrow \text{entier}$?

V-2 Exercice : Racines d'une quadratique avec une fonction

- Reprendre l'exemple du calcul des racines d'une équation de seconde degré et placer les calculs dans la fonction `quadratique(a, b, c)` qui reçoit en paramètre les coefficients a, b et c et renvoie les racines de cette équation.

☞ Remarques :

- Une fonction Python peut renvoyer différents types de données : il n'y a pas de contrôle (n'en déplaie aux *Camliens*!).
- Dans le cas présent, on décide de renvoyer un string (au lieu de float ou string).
- Savoir quel type de donnée une fonction doit renvoyer est davantage nécessaire si ce résultat doit participer à une expressions (P. ex. une expression effectuant \int ou \sum sur la fonction quadratique).
- Pour répondre à de telles situations en Python, le mécanisme d'*exception* permet de définir des *fonctions partielles* (comme `quadratique`) qui s'appliquent sous certaines conditions, voir la section [XV](#) page [27](#).

V-3 Exercice : Moyenne et écart type

- Dans cet exercice, on utilisera des fonctions travaillant avec une liste remplie par un tirage *Gaussien*.
 Pour ce faire, on écrira (pour le moment et avant d'en voir les détails) :

```
import random
echantillon=[random.gauss(16,2) for n in range(100)]
```

qui crée une liste de 100 nombres de la distribution $\mathcal{N}(16, 2)$.

- Écrire la fonction `moyenne` comme ci-dessus et utiliser cette fonction pour calculer *l'écart type* via la variance = $\text{moyenne}((x_i - \mu)^2)$ d'une suite de nombres telle que la liste `echantillon` ci-dessus.

→ Vous devez donc appeler `moyenne` avec une liste qui contient les $(x_i - \mu)^2$, $x_i \in \text{echantillon}$.

☞ Naturellement, vos résultats doivent (plus ou moins) s'accorder avec $\mathcal{N}(16, 2)$.

- Voir les *tirages aléatoires* en section [XVI](#), page [30](#) et les *listes en compréhension* en section [XXIII](#), page [47](#).

V-4 Variante : Exercice (suite écart type)

- Il y a une seconde manière de calculer l'écart type (qui est adaptée à un calcul de la moyenne et de la variance partielles, par exemple, lorsque les x_i se présentent au fur et à mesure) :

$$var = \frac{1}{N} \sum_1^N (x_i - \mu)^2 = \frac{1}{N} \left(\sum_1^N x_i^2 + \sum_1^N \mu^2 - \sum_1^N 2x_i\mu \right) = \frac{1}{N} \left(\sum_1^N x_i^2 + N\mu^2 - 2\mu \sum_1^N x_i \right) = \frac{1}{N} \left(\sum_1^N x_i^2 + N\mu^2 - 2\mu \cdot N \cdot \mu \right)$$

$$\text{Et donc } var = \frac{1}{N} \sum_1^N (x_i^2) - \mu^2.$$

- Il faut donc construire une liste contenant les x_i^2 , $x_i \in \text{echantillon}$.
- Écrire le code Python nécessaire pour calculer l'écart type de cette seconde manière (réutilisez `moyenne()` ci-dessus) et vérifier bien que le résultat correspond au calcul précédent.

VI Créer un script Python indépendant

- Supposons que nous souhaitons créer une version exécutable du calcul des racines d'une équation quadratique (vu ci-dessus) et de pouvoir utiliser cette fonction dans un programme indépendant (on veut un *script Python*).
→ Cela nous évitera de lancer `pyzo/spyder/...` pour utiliser notre fonction.

Pour commencer, on place le code vu ci-dessus dans le fichier `racines.py`. Pour les besoins de l'exemple, on simplifie le code en prenant moins de précautions au niveau des coefficients (car on maîtrise les arguments d'appel).

On n'oublie pas de tester cette fonction pour s'assurer qu'elle fonctionne !

```
from math import sqrt;

# résolution équation quadratique
def trinome(a, b, c):
    delta = b**2 - 4*a*c
    if delta > 0.0:
        assert(a != 0)
        racine_delta = sqrt(delta)
        return (2, (-b-racine_delta)/(2*a), (-b+racine_delta)/(2*a))
    elif delta < 0.0: return (0,)
    else:
        assert(a != 0)
        return (1, (-b/(2*a)))

# — Partie Principale —
print(trinome(1.0, -3.0, 2.0))
print(trinome(1.0, -2.0, 1.0))
print(trinome(1.0, 1.0, 1.0))

""" On obtient
(2, 1.0, 2.0)
(1, 1.0)
(0,) """
```

VI-1 Une façon simple

- Placer ce code dans un fichier `racines.py`.
- D'emblée, vous pouvez prendre une fenêtre "terminal" sur votre machine et taper la commande
\$ `python racines.py` (ou `c:\> python.exe racines.py` sous Windows)

→ La partie "principale" de votre code s'exécute et utilisera la fonction que vous avez définie.

- Si vous souhaitez éviter de taper `python` au début de la commande, vous pouvez :

- Sous linux : Ajouter au tout début de ce fichier la ligne (dite *shebang*)

```
#!/usr/bin/python ou (mieux) #!/usr/bin/env python
```

Puis rendez ce code exécutable par la commande Linux `chmod u+x racines.py`

Puis exécuter ce programme en tapant simplement dans une console \$ `./racines.py`

- Sous Windows : exécuter votre code dans une fenêtre de commande par `python.exe racines.py`
Windows ne semble pas avoir le mécanisme *shebang* (à moins d'avoir installé *cygwin*)

- Complétons en nous donnant les moyens d'importer nos propres fonctions pour les utiliser dans d'autres codes.

VI-2 Une solution plus élaborée

- Il arrive que l'on ait besoin d'utiliser les fonctions qu'on a définies dans un autre programme Python (à l'aide de `import`).
- On crée le fichier `utiliser_racines.py` avec le contenu suivant (placer les deux fichiers ".py" dans le même répertoire) :

```
from racines import trinome;
print(trinome(2.0, -4.0, 2.0))
```

- Puis exécuter ce dernier et obtenir (un peu trop de réponses) :

```
(2, 1.0, 2.0)      ← Qu'est-ce que c'est ?
(1, 1.0)           ← Et ça
(0,)               ← ;-)
(1, 1.0)           # Il nous faut juste ceci
```

- Le problème est que la partie principale de `racines.py` s'exécute aussi ! Et ce n'est pas ce que l'on veut.
- Résumons :
 - Nous voulons être capables de tester nos fonctions placées dans `racines.py`,
 - Nous voulons pouvoir lancer ce fichier comme un programme exécutable indépendant,
 - Nous voulons également réutiliser les dites fonctions dans d'autres programmes Python (comme nous l'avons fait dans `utiliser_racines.py`

- La solution est de faire précéder la partie "programme principal" de `racines.py` par :

```
if __name__ == "__main__":
```

Puis placer le contenu de la partie principale en dessous (avec indentation !)

- Cela donne ("assert" oubliés car on sait comment on appelle cette fonction depuis "main") :

```
from math import sqrt;

# fonction
def trinome(a, b, c):
    delta = b**2 - 4*a*c
    if delta > 0.0:
        racine_delta = sqrt(delta)
        return (2, (-b-racine_delta)/(2*a), (-b+racine_delta)/(2*a))
    elif delta < 0.0: return (0,)
    else: return (1, (-b/(2*a)))

if __name__ == "__main__":
    print(trinome(1.0, -3.0, 2.0))
    print(trinome(1.0, -2.0, 1.0))
    print(trinome(1.0, 1.0, 1.0))
```

- Pour exécuter ce programme, dans une fenêtre de console, taper" :

```
racines.py
```

OU (sous Windows) :

```
python racines.py
```

- On peut maintenant importer notre fonction dans `utiliser_racines.py` ci-après sans que la partie principale de `racines.py` ne s'exécute :

```
from racines import trinome;
print(trinome(2.0, -4.0, 2.0))
```

Puis exécuter ce programme (par les moyens expliqués ci-dessus) et obtenir :

```
(1, 1.0)
```

- Noter que `racines.py` reste indépendant et exécutable si on le souhaite.

VII Types principaux en Python

- Les types principaux de Python :

- **int** : sur 4 octes
- **long** : sur 4 (ou 8 octes)
- **float** : réel
- **str** : chaîne de caractères
- **bool** : booléen
- **tuple** : comme (1, 2, "ECL", 3.14)
- **list** : liste
- **set** : ensemble
- **range** : liste de valeurs à générer (xrange() en Python 2)
- **dict** : dictionnaire comme {'small' : 1, 'large' : 2}
- **complex** : nombre complexe (ex : 1j est une des racine de -1)
- **file** : fichier
- **none** : vide (équivalent à void)
- **exception** : exception
- **function** : fonction
- **module** : module
- **object** : objet
- **slice** : objet extensible
- **unicode** : chaîne de la norme Unicode
- **basestring** : str + Unicode

☞ La lettre 'j' ou 'J' est utilisée pour la partie imaginaire d'un nombre complexe. Pourquoi pas 'i' ou 'I' ? : les concepteurs de Python ont suivi les symboles utilisés en Ingénierie où 'I' est déjà utilisé comme symbole du courant.

☞ Voir ci-après pour les conversions entre ces types.

VII-1 Exemples

```
if type(1) is not int : print('oops')
if type('1') is not int : print(1)

type(3.14)
# float

type(None)
# NoneType

f = lambda c : c if type(c) is str and c.isalpha() else '?'
type(f)
# function

# Utilisation de f (voir plus loin les "lambda expressions") :
f('a') # donne 'a' car 'a' est alphabétique
f('1') # donne '?' car '1' n'est pas alphabétique
f(1)   # donne '?' car 1 n'est pas une chaîne
```

- Pour connaître les détails des types et leurs limites (les tailles sont en byte = 2 octets en Python) :

```
import sys;

print(sys.int_info)
# sys.int_info(bits_per_digit=30, sizeof_digit=4)

print(sys.float_info)
# sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308,
# min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)

print(sys.getsizeof(float)) # La taille de la classe float
# 400
print(sys.getsizeof(float())) # La taille d'un float
# 24 # Ce que coute en octes un réel (valeur + infos supplémentaires)
```

VIII Conversions de types

- Quelques conversions entre les types de données :

- int(val) : convertir en entier (la partie entière).
- long(val) : transforme une valeur en long.
- float(val) : transformation en flottant.
- complex(re, im) : transformation en nombre complexe.
- str(val) : transforme la plupart des variables en chaînes de caractères.

- `repr(val)` : similaire à `str`.
- `eval(str)` : évalue le contenu de son argument comme si c'était du code Python (déconseillé).
- `unicode(val)` : convertit en Unicode comme dans `u"aà eéèê"` # donne le code de la chaîne

VIII-1 Exemples de conversion de types

- caractères en string : liste de caractères entiers en string :

```
list1 = ['1', '2', '3']
str1 = ''.join(list1)

str1
# '123'
```

Ici, la fonction `join()` va accoler les éléments de son paramètre et les mettre à la que *leu-leu*. La chaîne vide (`"`) placée à gauche de `join()` sera le séparateur à placer entre les éléments accolés (ici rien).

- entiers en string : liste d'entiers en string : on convertit chaque entier en caractère (car `join()` s'applique aux strings) puis

```
list1 = [1, 2, 3]
str1 = ''.join(str(e) for e in list1) # ou avec "" à la place de ''

str1
# '123'
```

- string en entier : pour convertir un string `S` composé de chiffres en un entier, `int(S)` suffit. Par contre, si on n'est pas certain que les caractères de `S` sont des chiffres, un contrôle est nécessaire (voir exercice ci-dessous).
- string en réel : pour convertir un string `S` composé de chiffres en un réel, `float(S)` suffit. Cependant, la même précaution que pour les entiers ci-dessus s'applique.

```
a = "545.2222"
float(a)
# 545.222200000000004

int(float(a))
# 545
```

Ou en passant par une petite fonction (de même pour un entier) :

```
def is_float(value):
    try:
        float(value)
        return True
    except:
        return False
```

Pour les *exceptions*, voir la section [XV](#) en page [27](#).

VIII-2 Exercice

- Écrire la fonction `convertir(S)` qui convertit son paramètre (un string) en un entier. Pour que la conversion soit possible, le paramètre `S` doit être composé de chiffres éventuellement précédé d'un signe.

Noter qu'on peut accéder à `S[i]` (*i* commence à 0)

☞ Si un string `S` ne contient que des chiffres : `S.isdigit()` est vrai.

IX Modules Python

- Format général :

```
import tel_module
tel_module.fonc(..)    #appel de la fonction fonc du module
```

- Mieux conseillé :

```
import tel_module as mod1
import un_module as mod2
mod1.fonc(..)    #appel de la fonction fonc du tel_module
mod2.fonc(..)    #appel de la fonction homonyme du un_module
```

→ Cette écriture permet d'éviter les conflits de noms (cf. `fonc()`).

- On peut adapter un nom de fonction dans notre code :

```
import tel_module

def ma_fonction():
    localname = tel_module.fonc
    foo = localname(bar)
```

- Si on veut importer une fonction d'un module de façon plus précise :

```
from tel_module import fonc
fonc(..)    #appel de la fonction fonc du module
```

- Déconseillé : on risque de créer des conflits de noms :

```
from tel_module import *
fonc(..)    #appel de la fonction fonc du module
```

- Sachant qu'un module peut importer d'autres modules, préférez :

```
import scipy.stats as st
st.nanmean()
```

à cette autre manière équivalente :

```
import scipy.stats
scipy.stats.nanmean()
```

- On privilégie la parcimonie (importer seulement ce qui est nécessaire) :

```
from scipy.stat import nanmean, nanstd
nanmean(..)
nanstd()
```

- Un compromis intéressant : on peut renommer les fonctions importées pour éviter les conflits de noms :

```
from sys import argv as sys_argv
```

X La fonction range

- Format de la fonction `range()` :

`range([début], fin, [incrément])`

→ Le *début* et l'*incrément* sont optionnels (mis entre []).

→ Si on donne deux arguments à `range()`, ce seront alors le début et la fin de l'intervalle (et l'*incrément*=1).

Exemple : utilisez la fonction **`range()`** pour afficher une liste contenant

- les entiers de 0 à 3 ;
- les entiers de 4 à 7 ;
- les entiers de 2 à 8 par pas de 2

```
print("range(4) =", list(range(4)))
print("range(4, 8) =", list(range(4, 8)))
print("range(2, 9, 2) =", list(range(2, 9, 2)), "\n")
```

☞ Remarques : Python 3.4 "oblige" à écrire `print(list(range(4)))` au lieu de `print(range(4))` car `range(4)` ne représente pas l'intervalle souhaité sous forme de 0..3 (on dit qu'elle représente l'intervalle *par extension* ou *compréhension* alors que 0..3 est l'intervalle par extension obtenu via la fonction `list`)

```
R=range(4)
type(R)
# range

list(R)
# [0, 1, 2, 3]
```

X-1 Deux ou trois trucs pratiques sur range

1- Valeurs pour une séquence de variables :

```
a, b, c, d, e, f, g, h = range(8)
```

Permet d'assigner les valeurs 0..7 à la séquence a, b, c, d, e, f, g, h.

2- Valeurs d'un range() à l'envers :

```
range(4)[::-1]      # représente l'intervall à l'envers
list(range(4)[::-1]) # à l'envers
# [3, 2, 1, 0]
```

On obtient le même effet en explicitant l'intervalle [3..-1[et un *incrément* = -1 :

```
list(range(3,-1,-1)) # Noter les bornes
# [3, 2, 1, 0]
```

3- Interroger les valeurs potentielles d'un range :

```
R=range(4)      # R est du type range
R.count(0)      # Combien de '0' dans R
# 1
R.index(2)      # L'indice de '2' dans R
# 2
```

4- Dans range(debut, fin), on a en général $debut < fin$ tandis qu'avec $debut \geq fin$, on désigne un intervalle vide (et non une erreur).

```
list(range(1,1))
# []
```

X-2 Bonus : sommes

- Écrire le code Python en utilisant range() pour montrer les équivalences suivantes.
- En Python, $x \times \text{sum}(S) == \text{sum}(x \times y \text{ for } y \text{ in } S)$ représente $x \cdot \sum_{y \in S} y = \sum_{y \in S} x \cdot y$
- $\sum_{i=m}^n f(i)$ donne en Python $\text{sum}(f(i) \text{ for } i \text{ in range}(m, n+1))$ qui est équivalent à :

```
s = 0
for i in range(m, n+1):
    s += f(i)
```

- Les constantes multiplicatifs peuvent être insérées ou être sortie des sommes :

$$c \cdot \sum_{i=m}^n f(i) = \sum_{i=m}^n c \cdot f(i) \quad \text{car} \quad c(f(m) + \dots + f(n)) = c \cdot f(m) + \dots + c \cdot f(n).$$

- Associativité : sachant que $\sum_{i=m}^n f(i) + \sum_{i=m}^n g(i) = \sum_{i=m}^n (f(i) + g(i))$, on peut écrire en Python :
 $\text{sum}(f(i) \text{ for } i \text{ in } S) + \text{sum}(g(i) \text{ for } i \text{ in } S)$
 qui est la même chose que $\text{sum}(f(i) + g(i) \text{ for } i \text{ in } S)$.

→ Notons que ces équivalences ne tiennent que si les fonctions en questions n'ont pas d'effet de bord (ne modifient pas la mémoire).

- De même pour la soustraction.

XI Les listes

- Une introduction par des exemples de quelques fonctions sur les listes.
- Le premier élément d'une liste L est à l'indice 0, le dernier à l'indice $\text{len}(L) - 1$ ou à l'indice -1 et $L[-2] = L[\text{len}(L) - 2]$.
- Quelques fonctions (pour une liste L), voir les autres fonctions plus loin :

<code>L[:]</code>	toute la liste L (référence explicite aux éléments de L , cf. la copie d'une liste)
<code>L[k:p]</code>	$(k+1)$ ème et pème éléments (le premier élément à l'indice 0)
<code>L[:k]</code>	du début jusqu'au kème
<code>L[-1]</code>	dernier élément
<code>L[0]</code>	premier élément
<code>L[i]=val</code>	remplace $L[i]$ par val (si $L[i]$ existe, sinon exception <i>IndexError</i>).
<code>L=list(itérable)</code>	crée une liste à partir d'un itérable (comme <i>range()</i>), une liste est elle-même itérable.
<code>len(L)</code>	nombre d'éléments de L
<code>L.append(val)</code>	ajout d'un élément (à la fin, L est modifiée)
<code>L.remove(val)</code>	suppression de la première occurrence de val (L est modifiée)
<code>L.reverse()</code>	renverse L (L est modifiée)
<code>L.index(val)</code>	renvoie l'indice de la 1ère occurrence de val dans L , erreur si val pas dans L .
<code>L.copy()</code>	copie le contenu de L
<code>L.count(val)</code>	compte le nombre d'occurrence de val dans L
<code>L.extend(itérable)</code>	ajoute à L les éléments donnés par itérable, équivalent à <code>list[len(list):] = L</code>
<code>del(L[i])</code> ou <code>del D[i]</code>	effacement de l'élément d'indice i d'une liste (L est modifiée)
<code>L.clear()</code>	efface le contenu de L . Équivalent à <code>del L[:]</code>
<code>L.insert(i, x)</code>	insertion de x <u>avant</u> l'indexe i <code>L.insert(0, x)</code> insère x au début de L et <code>L.insert(len(L), x)</code> est équivalent à <code>L.append(x)</code> .
<code>L.pop()</code>	supprime et renvoie le dernier élément.
<code>L.pop(i)</code>	supprime un élément en position i et renvoie cet élément.

- Deux fonctions pour le tri d'une liste :

<code>L.sort()</code>	trie la liste L (L est modifiée)
<code>sorted(L)</code>	trie la liste L (voir exemples plus loin)

Remarques et rappels sur **Indices et Listes**

- Dans une liste, les indices négatifs commencent par la fin :
-1 : le dernier, -2 : l'avant dernier ...
- $L[0:5]$ est la même chose que $L[:5]$: du 1er au 5e inclus ($[0..5[$)
- Plus généralement : du $k_{\text{ème}}$ au $p_{\text{ème}}$ élément s'écrira $L[k-1:p]$ (le premier à l'indice 0)
Si $L=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, pour avoir du 3ème au 5ème, on écrit $L[2:5]$
- Si $L=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, L à l'envers : $L[::-1] = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]$.
- pour $L[0:10] \rightarrow [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, les indices d'une tranche (*slice*) peuvent être négatifs
 $x[-5:-2] \rightarrow [5, 6, 7]$
 $x[-1] \rightarrow 9$
- Une tranche (*slice*) : *une_liste[début : fin : incrément]*
 $R=\text{range}(10)$ # R vaut (par *intention*) $\rightarrow [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
 $\text{list}(R[1:7:2])$ # la tranche $[1..7[$ d'incrément 2
$\rightarrow [1, 3, 5]$
- Tranche avec incrément : pour $L[0:10] \rightarrow [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
 $x[0:10:2] \rightarrow [0, 2, 4, 6, 8]$ l'incrément est à la fin (par opp. à Matlab : au milieu)
Et à l'envers : $L[8:0:-2] \rightarrow [8, 6, 4, 2]$
- Si l'intervalle est limité par un bord, on peut l'omettre : pour $L[0:10] \rightarrow [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
 $L[:] \rightarrow [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
 $L[:2] \rightarrow [0, 2, 4, 6, 8]$
 $L[-5:] \rightarrow [5, 6, 7, 8, 9]$
 $L[8::-2] \rightarrow [8, 6, 4, 2, 0]$

Exemple d'affectation par tranches

```
L = [1, 2, 3, 4, 5]
L[2:3] = [0, 0] # place '0' à l'indice 2 et ajoute tout le reste à la suite de celui-ci.

L
# [1, 2, 0, 0, 4, 5]

L[1:1] = [8, 9] # Idem
L
# [1, 8, 9, 2, 0, 0, 4, 5]

L[1:-1] = [] # élément d'indice 1 au dernier (mais le dernier exclu) devient "vide"
L
# [1, 5]
```

XI-1 Un exemple sur les listes

- Définir la liste `liste = [17, 38, 10, 25, 72]`, puis effectuez les actions suivantes :
 - triez et affichez la liste ;
 - ajoutez l'élément 12 à la liste et affichez la liste ;
 - renversez et affichez la liste ;
 - affichez l'indice de l'élément 17 ;
 - enlevez l'élément 38 et affichez la liste ;
 - affichez la sous-liste du 2e au 3e élément ;
 - affichez la sous-liste du début au 2e élément ;
 - affichez la sous-liste du 3e élément à la fin de la liste ;
 - affichez la sous-liste complète de la liste ;
 - affichez le dernier élément en utilisant un indicage négatif.
 - affichez l'avant-dernier élément en utilisant un indicage négatif.
 - puis définir L comme une liste des entiers de 0 à 5 et testez l'appartenance des éléments 3 et 6 à L.

☞ Bien remarquer que certaines méthodes de liste ne retournent rien.

```
nombres = [17, 38, 10, 25, 72]
print(" Liste initiale ".center(50, '-'))
print(nombres, '\n')

print(" Tri ".center(50, '-'))
nombres.sort() # TRI sur place (la liste est modifiée)
print(nombres, '\n')

print(" Ajout d'un element ".center(50, '-'))
nombres.append(12)
print(nombres, '\n')

print(" Retournement ".center(50, '-'))
nombres.reverse()
print(nombres, '\n')

print(" Indice d'un element ".center(50, '-'))
print(nombres.index(17), '\n')

print(" Retrait d'un element ".center(50, '-'))
nombres.remove(38)
print(nombres, '\n')

print(" Indicage ".center(50, '-'))
print("nombres[1:3] =", nombres[1:3])
print("nombres[:2] =", nombres[:2])
print("nombres[2:] =", nombres[2:])
print("nombres[:] =", nombres[:])
print("nombres[-1] =", nombres[-1])
print("nombres[-2] =", nombres[-2])

L = list(range(6)) # range(6) représente 0..5 qu'on transforme en une liste
print("La liste =", L)
print("Test d'appartenance de l'element 3 :", 3 in L)
print("Test d'appartenance de l'element 6 :", 6 in L)
```

XI-2 Autres exemples de fonctions sur les listes

- Retrait du dernier élément (fonction partielle) avec `pop()` :

```
L=[1,3,2]
L
# [1,3,2]
L.pop()
# 2
L
# [1,3]
L.pop(); L.pop()
# 1
L
# []
L.pop()
# IndexError: pop from empty list
```

☞ `L.pop(i)` enlève et renvoie l'élément d'indice `i` (à partir de 0) de la liste `L`.

Si `i > L.len()-1`, on aura une exception `IndexError: pop index out of range`.

- Copie de liste :

```
L=[1,2,3]
L1=L #L1 est homonyme de L; ce n'est pas une copie (la liste a maintenant deux noms !)
L[0]=6
L
# [6, 2, 3]
L1
# [6, 2, 3]

#Pour copier une liste, il faut être plus explicite :
L2=L[:] # copier vraiment L dans L2
L[1]=7
L
# [6,7, 3]
L1
# [6,7, 3]
L2
# [6,2, 3]
```

→ L'équivalent en *numpy* de cette copie : `L2=L.copy()`

☞ On dit qu'en Python, une variable est avant tout son contenu et le nom n'est qu'un attribut de la variable. C'est pour cette raison qu'on doit copier le contenu. Cette différence reflète le comportement des *copie-constructeurs*.

→ N'oublions pas que ce comportement (deux noms font référence au même objet) s'applique aux objets mutables (*List, Dict, Set, etc.*), pas aux non mutables (comme les entiers, constantes strings, tuples, None).

Pour éviter cette confusion, les fonctions de la librairie `std` de Python rendent `None` lorsqu'elles modifient un objet mutable (cf, `append`, `sort`). Notons aussi que cette confusion n'arrive pas en cas de création d'un nouvel objet tel que `y=y+[10]` (où `y` : un entier) ou `sorted(liste)`.

Il y a une exception à cette règle : l'opérateur `+=`. Dans `liste += [1, 2, 3]`, on applique en fait la fonction `liste.extend([1, 2, 3])` à l'objet mutable `liste` tandis que `entier_y += 1` ou `tuple1 += (1, 2, 3)` créent de nouvelles variables car les entiers et les tuples ne sont pas mutables.

Enfin, pour savoir si deux variables font référence à un même objet, utiliser `is` ou la fonction `id()`.

- **`sorted()` et `sort()`** (Python 3.4, noter le tri à l' **envers** :

```
sorted([5, 2, 3, 1, 4])    # Crée une nouvelle liste
# [1, 2, 3, 4, 5]

sorted([5, 2, 3, 1, 4], reverse=True)    # Crée une nouvelle liste. On trie à l'envers
# [5, 4, 3, 2, 1]

a = [5, 2, 3, 1, 4]
a.sort()    # Tri sur place (la liste est modifiée)
a
# [1, 2, 3, 4, 5]

sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
# [1, 2, 3, 4, 5]
```

☞ Remarques : `sort()` ne s'applique qu'aux listes alors que `sorted()` s'applique à tout ce qui "itérable" (y compris les *Dicos*).

- `sorted()` et `sort()` acceptent un paramètre *key* qu'on peut utiliser pour préparer chaque élément avant le tri.

```
#Dans cet exemple, on demande (par split()) à isoler les mots puis à trier leur forme minuscule

sorted("Ceci est une Chaîne de caractères Debase!".split(), key=str.lower) #passer d'abord en minuscules
# ['caractères', 'Ceci', 'Chaîne', 'de', 'Debase!', 'est', 'une']

# Nom, Note et Age des étudiants
student_tuples = [
    ('john', 'A', 25),
    ('pasSa', 'B', 22),
    ('tronche', 'B', 20),
]

sorted(student_tuples, key=lambda student: student[2])    # sort suivant l'Age
#[('tronche', 'B', 20), ('pasSa', 'B', 22), ('john', 'A', 25)]
```

☞ Remarques : la liste `student_tuples` contient des tuples (ici des triplets). Voir plus loin pour les tuples la section [XII page 19](#).

Ici, on peut aussi utiliser `sort()`.

XI-3 Itération à l'envers

```
for x in reversed(sequence):
    ... # do something with x...
```

- Par exemple :

```
S=[1,2,3,4]
for x in reversed(S):
    print(x)
"""
4
3
2
1
"""
```

- Ou par :

```
S=[1,2,3,4]
for x in S[::-1]:
    print(x)
"""
4
3
2
1
"""
```

XI-4 Itération sur une liste avec enumerate

- Si on doit écrire les éléments d'une liste avec le rang de chacun :

```
strings = ['a', 'b', 'c', 'd', 'e']
for index, string in enumerate(strings):
    print(index, string)

"""
0 a
1 b
2 c
3 d
4 e """
```

- On peut réaliser le même effet avec un accès aux éléments par indice :

```
S = ['a', 'b', 'c', 'd', 'e']
for index in range(len(S)):
    print(index, S[index])

"""
0 a
1 b
2 c
3 d
4 e """
```

XI-5 Exemple : Pile

- Implantez une pile LIFO avec une liste.

☞ Remarques : dans une spécification formelle (appelée TDA : type de données abstrait) des piles, on utilise une notation algébrique pour spécifier le type Pile. On précise d'abord la *signature* des opérateurs. Deux autres parties *préconditions* et *axiomes* viennent compléter la définitions d'un TDA.

- Ce qui est important est que chaque fonction renvoie une et une seule valeur et que les fonctions *partielles* (celles avec le symbole \rightarrow) ne s'appliquent que sous conditions (exprimées dans la partie *préconditions*).
- Par exemple, pour une pile, on aura :

type : pile ce que ce TDA définit
utilise : Bool, Data, Nat les types que ce TDA utilise (une sorte de "import")

Signature :

vide :	\rightarrow pile	la pile vide (une constante = fonction 0-aire=sans argument)
empile :	pile \times Data \rightarrow pile	ajout d'un élément
depile :	pile \rightarrow pile	fonction partielle (non définie si pile vide). Noter qu'on ne renvoie pas un Data
top :	pile \rightarrow Data	consultation du 1er élément. Fonction partielle (non définie si la pile est vide)
est_vide :	pile \rightarrow Bool	test de vacuité
len :	pile \rightarrow Nat	nbr d'éléments

Préconditions :

ici, on précise les conditions d'applications des fonctions partielles

Par exemple : `depile(p) : not (est_vide(p)) ...`

Axiomes (les propriétés des fonctions) :

Par exemple : pour $e : \text{data}$, $p : \text{pile}$

```
est_vide(vide)=True
est_vide(empiler(p,e))=False
depile(empiler(p,e))=p
top(empiler(p,e))=e
....
```

- Noter que le concept de classes convient parfaitement aux objets Piles (voir en TC2).

- Une implantation partielle à titre d'exemple :

```

def empile(p, a):
    p.append(a)

def depile(p):
    try:
        p.pop()
        return p
    except:
        print("La pile est vide !")
        return None

def top(p):
    try:
        return p[0]
    except:
        print("La pile est vide !")
        return None

def est_vide(p):
    return p==[]

print(" Pile initiale ".center(50, '-'))
print(" Empilage ".center(50, '-'))
empile(lifo, 5); empile(lifo, 8); empile(lifo, 11)
print("lifo :", lifo, '\n')
rien = input("Entree")

print(" Empilage ".center(50, '-'))
empile(lifo, 11)
print("lifo :", lifo, '\n')
rien = input("Entree")

print(" Depilages ".center(50, '-'))
for i in range(5):          du premier au 5e (d'indice 0 à l'indice 4)
    depile(lifo)
    print("lifo :", lifo)

""" TRACE
_____ Pile initiale _____
lifo : [5, 8, 9]

"Entree"
_____ Empilage _____
lifo : [5, 8, 9, 11]

Entree"
_____ Depilages _____
lifo : [5, 8, 9]
lifo : [5, 8]
lifo : [5]
lifo : []
La pile est vide !
lifo : []
"""

```

XI-6 Bonus : TDA Queue

- De la même manière, implémentez une queue FIFO avec une liste.

XI-7 Exercice : enlever les doublons d'une liste

- Supprimer les éléments répétés dans une liste sans changer l'ordre des éléments.
- Notons qu'en Python, pour une liste $L=[5,1,2,8,1,5]$, le même résultat est obtenu par `set(L)` qui transforme L en un ensemble. Par contre, l'ensemble obtenu sera ordonné. C'est ce que l'on veut éviter ici.

XI-8 Bonus : Scinder une liste

- Échantillonner une liste L d'entiers de taille $N > 10$ au hasard (entre $2..N * 4$) puis scinder cette liste en L_1, L_2 :
 L_1 contiendra les sous-séquences ascendantes de L de longueur d'au moins 2, L_2 les autres.
- Exemple-1 : si $L = [27, 6, 32, 13, 28, 12, 34, 37, 2, 30]$ alors $L_1 = [6, 32, 13, 28, 12, 34, 37, 2, 30]$, $L_2 = [27]$
- Exemple-2 : Si $L = [18, 17, 27, 18, 40, 22, 30, 30, 22, 15]$, on aurait $L_1 = [17, 27, 18, 40, 22, 30, 30]$, $L_2 = [18, 22, 15]$

XII Tuples

- Le tuple vide se note : ()
- On accède au range d'un élément d'un tuple par la fonction *index*.
- On teste l'appartenance d'un élément à un tuple par *in*.
- Si les (éléments des) tuples sont des objets non mutables, on ne peut pas les modifier.
- Exemple :

```
a_tuple = (1, 2)
a_tuple += (3,4)
a_tuple
# (1, 2, 3, 4)

# a_tuple[0] += 3 : erreur : affectation / modification interdites

a_tuple[3]
# 4
```

- Tentative de modification des éléments non mutables du tuple :

```
a_tuple[4]=5      # --> erreur affectation / modification interdites
a_tuple += (5)    # --> erreur can only concatenate tuple (not "int") to tuple
```

→ Remarquer les 2 erreurs.

- Mais si un élément du tuple est mutable (ici une liste), alors les modifications sont possibles :

```
a_tuple += (['Hello'], 'Paris')
a_tuple
# (1, 2, 3, 4, ['Hello'], 'Paris')

a_tuple.index(['Hello'])
# 4

# a_tuple[4] = ['GoodBye'] : erreur affectation / modification interdites
```

→ Le tuple est allergique à l'opérateur d'affectation.

- Par contre, on peut modifier le contenu de la liste ['Hello']

```
a_tuple[4].append('World')
a_tuple
# (1, 2, 3, 4, ['Hello', 'World'], 'Paris')

a_tuple[4]+=['of Lions']
a_tuple
# (1, 2, 3, 4, ['Hello', 'World', 'of Lions'], 'Paris')
```

- On peut également modifier un élément de la liste ['Hello', 'World', 'of Lions']

```
a_tuple[4][0]= 'God Luck'
a_tuple
# (1, 2, 3, 4, ['God Luck', 'World', 'of Lions'], 'Paris')
```

- On peut ajouter des doublets ou plus à un tuple. On va ajouter un couple puis un triplet :

```
a_tuple += (5, 'toto')
a_tuple
# (1, 2, 3, 4, ['God Luck', 'World', 'of Lions'], 'Paris', 5, 'toto')

a_tuple += (10,20,30)
a_tuple
# (1, 2, 3, 4, ['God Luck', 'World', 'of Lions'], 'Paris', 5, 'toto', 10, 20, 30)
```

XII-1 Accès aux éléments d'un tuple

- Comme pour les liste, on peut avoir accès aux éléments du tuple par indice négatif et par tranches. Exemple :

```
t = ("a", "b", "c", "z", "example")
t
# ('a', 'b', 'c', 'z', 'example')

t[0]
# 'a'

t[-1]
# 'example'

t[1:3]
# ('b', 'c')

z in t
# True
```

- On peut associer des variables aux membres d'un tuple :

```
v = ('a', 'b', 'e') # v est un tuple
(x, y, z) = v
x
# 'a'

y
# 'b'

z
# 'e'
```

- Association d'un rang (à partir de 0) aux éléments d'un tuple.

```
list(range(7)) # 'list' nécessaire en Python 3
#[0, 1, 2, 3, 4, 5, 6]

(vertu, genie, travail, opinion, recompenses, revolution, sanculottides) = range(7)
vertu
# 0

sanculottides
# 6
```

- Mais aussi, il est possible d'associer un rang $\neq 0$ au premier élément. Noter la fonction *len* :

```
len((vertu, genie, travail, opinion, recompenses, revolution, sanculottides))
#7

(vertu, genie, travail, opinion, recompenses, revolution, sanculottides) = range(3, 10)
vertu
# 3
```

```
% a_tuple
%# (1, 2, 3, 4, ['God Luck', 'World', 'of Lions'], 'Paris', 10, 20, 30)
%
% print(*a_tuple)
%# 1 2 3 4 ['God Luck', 'World', 'of Lions'] Paris 10 20 30
%
```

XIII Introduction à la mise en page

- Écrire des choses avec `print` :

```
q = 459
p = 0.098
print(q, p, p * q)
# Donne —> 459 0.098 44.982

print(q, p, p * q, sep=",")
# Donne —> 459,0.098,44.982

print(q, p, p * q, sep=" :-) ")
# Donne —> 459 :-) 0.098 :-) 44.982
```

☞ `print` renvoie `None` (au cas où vous vous laisseriez aller à la programmation fonctionnelle!).

```
print(print(1))
# 1
# None
```

- On peut créer des strings et les écrire

```
print(str(q) + " >> " + str(p) + " << " + str(p * q))
# Donne —> 459 >> 0.098 << 44.982
```

- Le format : opérateur `"%"` (hérité du langage C)

```
x=5
'%d' % x
# Donne —> '5'

'(%d)' % x
# Donne —> '(5)'

'Je connais un mouton à %d pattes.' % x
# Donne —> 'Je connais un mouton à 5 pattes.'

y=2
'Je connais %d moutons à %d pattes.' % (y,x)
# Donne —> 'Je connais 2 moutons à 5 pattes.'
```

☞ Remarques : dans ces exemples, **"%" sépare** le string exprimant le format (e.g. "%d") de la valeur à écrire (e.g. "x").

- On peut maîtriser les longueurs des champs (e.g. pour justifier des colonnes) :

→ La syntaxe à suivre est **%[flag][longueur][.precision] variable**

```
r=2
print("La surface pour un rayon <%3d> sera <%8.3f>" % (r, r*r*pi))
# Donne → La surface pour un rayon < 2> sera < 12.566>
```

- Ici, %3d demande à écrire r sur 3 positions (on écrit entre <> pour plus de clarté). Pour la surface, %8.3f demande une longueur totale de 8 places (le '.' compris) dont 3 chiffres après la virgule. En cas d'erreur de notre part pour la totalité de la longueur, par exemple si on demande <%1.3f>, Python outrepassa l'erreur et utilise ce qui est juste (ici 3 chiffres après la virgule), sinon le minimum de longueur requise.

Par exemple, si on demande %4.2f pour la surface, les 2 chiffres après la virgule demandés plus une place nécessaire pour le '.' feront que le total de 4 places ne suffiront pas (il faut au moins 2 places pour 12). Au final, on écrira <12.57>.

- Noter que %3d est équivalent à %3i.
- Quelques informations supplémentaires (extrait de la doc) :

Symbole de conversion (suit %)	Signification
d	entier décimal signé.
i	entier décimal signé.
o	octal Non signé .
u	décimal Non signé .
x	hexadécimal non signé (minuscule).
X	hexadécimal non signé (majuscule).
e	exponentiel au format virgule flottante (minuscule).
E	exponentiel au format virgule flottante (majuscule).
f	décimal au format virgule flottante.
F	décimal au format virgule flottante.
g	Comme "e" si exposant > -4 ou < à la précision, "f" sinon.
G	Comme "E" si exposant > -4 ou < à la précision, "F" sinon.
c	Un caractère (ou un entier ou un string mono caractère).
r	String (convertit tout objet Python à l'aide de repr()).
s	String (convertit tout objet Python à l'aide de str()).
%	Pas de conversion, produit le caractère "%".

- Quelques exemples :

```
print("%10.3e"% (356.08977))
#Donne → 3.561e+02
print("%10.3E"% (356.08977))
#Donne → 3.561E+02
print("%10o"% (25))
#Donne → 31
print("%10.3o"% (25))
#Donne → 031
print("%10.5o"% (25))
#Donne → 00031
print("%5x"% (47))
#Donne → 2f
print("%5.4x"% (47))
#Donne → 002f
print("%5.4X"% (47))
#Donne → 002F
print("Only one percentage sign: %%" % ())
#Donne → Only one percentage sign: %
print('%d' % 0x12)
#Donne → 18
```

XIII-1 La fonction format

- La section ci-dessus (avec "%") est un héritage du langage C. Ce mécanisme peut être insuffisant dans certain cas, par exemple pour écrire une liste (à moins de l'écrire élément par élément ou de la transformer par `repr()`).
- On peut utiliser un formatage (à la Python) à l'aide de la fonction **format**. Dans :

```
r=2
"La surface pour un rayon {0:3d} sera {1:8.3f}".format(r, r*r*pi)
# Donne --> 'La surface pour un rayon 2 sera 12.566'
```

Le `{0:3d}` précise le format de la première colonne (commence à 0) et `{1:8.3f}` celui de la 2e colonne.

Noter la position de la fonction `format`. N'oubliez pas que `' % '` est absent dans l'expression de format (**ne pas écrire** `1:%8.3f`)

- D'autres exemples :

```
"First argument: {0}, second one: {1}".format(47,11)
# Donne --> 'First argument: 47, second one: 11'
"Second argument: {1}, first one: {0}".format(47,11)
# Donne --> 'Second argument: 11, first one: 47'
"Second argument: {1:3d}, first one: {0:7.2f}".format(47.42,11)
# Donne --> 'Second argument: 11, first one: 47.42'
"First argument: {}, second one: {}".format(47,11)
# Donne --> 'First argument: 47, second one: 11'
# arguments can be used more than once:
...
"various precisions: {0:6.2f} or {0:6.3f}".format(1.4148)
# Donne --> 'various precisions: 1.41 or 1.415'
```

- Aussi, on peut remplacer la numérotation des colonnes et renommer les argument. Dans l'exemple ci-dessous, le rayon représente *r* et la surface la *superficie*.

```
r=2
"La surface pour un rayon {rayon:3d} sera {surface:8.3f}".format(surface=r*r*pi, rayon=r)
# Donne --> 'La surface pour un rayon 2 sera 12.566'
```

- D'autres exemples (noter les longueurs des champs utilisées pour les strings).
- N.B. : le symbole `'<'` impose une justification à gauche (avec ajout d'espaces si nécessaires) et `'>'` une justification à droite.

```
print('%(language)s has %(number)03d quote types.' % {"language": "Python", "number": 2})
# Python has 002 quote types.

"{0:<20s} {1:6.2f}".format('Spam & Eggs:', 6.99)
# Donne --> 'Spam & Eggs: 6.99'

"{0:>20s} {1:6.2f}".format('Spam & Eggs:', 6.99)
# Donne --> 'Spam & Eggs: 6.99'

"{0:>20s} {1:6.2f}".format('Spam & Ham:', 7.99)
# Donne --> 'Spam & Ham: 7.99'

"{0:<20s} {1:6.2f}".format('Spam & Ham:', 7.99)
# Donne --> 'Spam & Ham: 7.99'

"{0:<20} {1:6.2f}".format('Spam & Ham:', 7.99)
# Donne --> 'Spam & Ham: 7.99'

"{0:>20} {1:6.2f}".format('Spam & Ham:', 7.99)
# Donne --> 'Spam & Ham: 7.99'
```

- Un exemple de mise en colonne de valeurs : afficher en colonnes de x, x^2, x^3 avec $x \in 1..10$

```
for x in range(1, 11):
    print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))

1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

# On peut aussi ne rien écrire entre les {}:
x,y='choux',1
print("Il n'y a que {} façon de planter des {} chez {}".format(y, x, 'nous'))
# Donne --> Il n'y a que 1 façon de planter des choux chez nous.
```

XIII-2 Exercice : table de x , x^2 , \sqrt{x}

- Écrire un programme qui affiche en colonnes la table contenant les colonnes x , x^2 et \sqrt{x} pour $x \in 1..N$. Lire au préalable la valeur de N (ou supposer $N = 20$).

XIV Itérations : While, for

- Il est difficile d'éviter de rencontrer les itérations (en particulier avec `for`) en Python et ce assez rapidement. Voyons quelques exemples moins triviaux.

XIV-1 While : Celcius-Fahrenheit

- Programmer une table de conversion Celcius-Fahrenheit et écrire les valeurs en colonnes.
- On utilisera les différentes formes d'itérations.

```
print("Table de conversion fahrenheit -> celsius")
print("-"*60)
inf,sup, pas=0,100,20;

# Première méthode : avec while et sans véritable mise en page
fahr=inf;
while(fahr<=sup):
    celsius = (5.0/9.0)*(fahr-32.0);
    print("En fahrenheit : ",fahr," - En celsius : ",celsius)
    fahr = fahr+ pas

""" TRACES
Table de conversion fahrenheit -> celsius
-----
En fahrenheit : 0 - En celsius : -17.77777777777778
En fahrenheit : 20 - En celsius : -6.666666666666667
En fahrenheit : 40 - En celsius : 4.444444444444445
En fahrenheit : 60 - En celsius : 15.555555555555557
En fahrenheit : 80 - En celsius : 26.666666666666668
En fahrenheit : 100 - En celsius : 37.77777777777778
"""

# Une variante de la première méthode : avec while + break
fahr=inf;
while True :
    celsius = (5.0/9.0)*(fahr-32.0);
    # Première façon de formater
    print("En fahrenheit : ",repr(fahr).rjust(5),
          " - En celsius : %8.3f" % celsius)
    fahr = fahr+ pas
    if (fahr>sup) :
        break

""" TRACE
En fahrenheit : 0 - En celsius : -17.778
En fahrenheit : 20 - En celsius : -6.667
En fahrenheit : 40 - En celsius : 4.444
En fahrenheit : 60 - En celsius : 15.556
En fahrenheit : 80 - En celsius : 26.667
En fahrenheit : 100 - En celsius : 37.778
"""

%\usepackage{setspace,listings} % setspace important
% on peut donner \begin{lstlisting}[caption=.. , label=...]
```

XIV-2 For : Celcius-Fahrenheit

- La même question avec `for` :

```
# Deuxième méthode : avec for
for fahr in range(inf, sup+pas, pas):
    # Deuxième façon de formater (pour les réels)
    celsius = (5.0/9.0)*(fahr-32.0);
    print("En fahrenheit : ",repr(fahr).rjust(5), " - En celsius : %8.3f" % celsius)

""" TRACE
En fahrenheit : 0 - En celsius : -17.778
En fahrenheit : 20 - En celsius : -6.667
En fahrenheit : 40 - En celsius : 4.444
En fahrenheit : 60 - En celsius : 15.556
En fahrenheit : 80 - En celsius : 26.667
En fahrenheit : 100 - En celsius : 37.778
"""

# Variante de for avec une autre mise en page
```



```

inf,sup, pas=0,100,20;
print("Fahre
nheit".center(10), "celsius".center(10), sep=' ')
print("-"*60)
fahr=inf;
nb_pas=sup//pas          # le nbr de pas
for indice in range(0, nb_pas+1):
    celsius = (5.0/9.0)*(indice*pas+fahr-32.0);

    # 0:xx et 1:yyy font références aux numéros des colonnes dans format
    # 0:xx fait référence au champ No 0 = 1ère colonne
    print( '{0:7d}          {1:4.1f}'.format(indice*pas+fahr, celsius))

"""
Fahrenheit      celsius
    0           -17.8
   20           -6.7
   40            4.4
   60           15.6
   80           26.7
  100           37.8
"""

```

→ Noter que dans `for i in range(5) : ...`, la valeur de `i` à la sortie de ce `for` est 4 (et non 5 comme on pourrait le penser).

☞ Remarques : le format de `for` :

```

for iteration_for in intervalle :
    corps_for
else : else_expression

```

- Noter que `intervalle` est évalué une seule fois et donne lieu à un itérateur dont les valeurs sont **assignées** une par une à `iteration_for` (donc, `c` doit pouvoir recevoir une valeur, ou être une *lvalue*, ce qui exclue par exemple un appel de fonction).

- `for` peut provoquer une exception "StopIteration".

- `else_expression` est exécutée à la fin des itérations ou suite à l'exception "StopIteration".

- `corps_for` peut contenir `break`.

Si `break` est exécuté, on abandonne l'itération sans exécuter `else_expression`.

- `corps_for` peut contenir `continue`. S'il est exécuté, on passe à l'itération suivante (mais s'il n'y aucune autre itération possible, `else_expression` sera exécutée avant de terminer).

- Les variables de `iteration_for` reçoivent des valeurs qui écrasent leur éventuelle ancienne valeur :

```

# Observer i

i=12
for i in range(5) :
    print(i, end=' ')
    i=42
print(i)
# Donne —> : 0 1 2 3 4 42

```

- On constate que `i=12` ni les `i=42` n'ont aucun effet sur `i` de `iteration_for`. Cependant, la valeur finale de `i=42` (fait à la toute dernière itération).

- On aura le même résultat avec (noter `else`) :

```

# Observer i

i=12
for i in range(5) :
    print(i, end=' ')
else : i=42

print(i)
# Donne —> : 0 1 2 3 4 42

```

- Les variables de `iteration_for` existeront après les itérations. Elles auront leur valeur finale de la dernière itération.

- Éviter de modifier `intervalle` dans `corps_for` sous peine de surprises.

☞ Le schéma `while` dispose également d'une clause `else : ...` dont le comportement est identique à celle présente dans `for`. Idem pour `continue` et `break` dans le schéma `while`.

XIV-3 Exercice : valeur de e

- Écrire un programme qui estime la valeur de la constante mathématique **e** en utilisant la formule : $e = \sum_{i=0}^n \frac{1}{i!}$

Pour ce faire, définissez la fonction *factorielle* puis, dans votre programme principal, lire au clavier l'ordre *n* et affichez l'approximation correspondante de *e*.

☞ Remarques : la fonction `math.factorial(.)` existe.

XIV-4 Exercice : nombre premier

- On appelle *nombre premier* tout entier naturel supérieur à 1 qui possède exactement deux diviseurs, lui-même et l'unité ;
- On dira donc que l'entier positif *N* est premier si $N = 1$ ou $N = 2$ ou alors *N* remplit les conditions suivantes :
 - $N \% 2 \neq 0$
 - $\sqrt{N} \notin \mathbb{N}$
 - $\nexists k, (k \in 3 \dots \lceil \sqrt{N} \rceil \wedge k | N)$ c-à-d. : *N* n'est multiple d'aucun des entiers $k \in 3.. \sqrt{N} + 1$
- Lire un entier et tester s'il s'agit d'un nombre premier.

XIV-5 Bonus : Nombre Premier par diviseurs propres

- Une autre méthode : on appelle *diviseur propre* de *N*, un diviseur quelconque de *N* (1 et *N* exclus) ;
- Lire *N* et compter le nombre de diviseurs propres du nombre *N*. S'il n'y a pas de tel diviseur alors *N* est premier. Sinon, on affiche ses diviseurs ainsi que leur nombre.

XIV-6 Bonus : Diviseurs des nombres Parfaits et Premiers

- Un entier naturel est dit *parfait* s'il est égal à la somme de tous ses diviseurs propres ;
- Les nombres *a* tels que : $(a + n + n^2)$ est *premier* pour tout *n* tel que $0 < n < (a - 1)$, sont appelés nombres *chanceux*.
- Écrire un script Python définissant quatre fonctions : *somDiv*, *estParfait*, *estPremier*, *estChanceux*.
 - la fonction *somDiv* retourne la somme des diviseurs propres de son argument ;
 - les trois autres fonctions vérifient la propriété donnée par leur définition retournant un booléen. Plus précisément, si par exemple la fonction *estPremier* vérifie que son argument est premier, elle retourne True, sinon elle retourne False.
- Écrire et tester ces fonctions, par exemple avec les valeurs *somDiv*(12), *estParfait*(6), *estPremier*(31) et *estChanceux*(11).

XIV-7 Bonus (suite) : Importer vos fonctions

- Transformer le module précédent en le nommant *parfait_chanceux_m.py* définissant les quatre fonctions : *somDiv*, *estParfait*, *estPremier*, *estChanceux*.
- Penser à la partie principale (voir la section VI, page 6) : pour transformer ce module en un script indépendant (et importer les fonctions qui y sont définies), vous devez ajouter avant la partie principale :


```
if __name__ == '__main__':
```
- Écrire le programme principal *parfait_chanceux.py* qui comporte :
 - l'initialisation de deux listes : parfaits et chanceux ;
 - une boucle de parcours de l'intervalle [2, 1000] incluant les tests nécessaires pour remplir ces listes ;

XIV-8 Exemple : compter les lettres

- On lit un texte terminé par un '.'. Compter le nombre de chiffres et de lettres.

```
ph=input("Donner une phrase sur une ligne terminée par un point ")
nb_chiffres=nb_lettres=0
for car in ph:
    if car=='.':
        break
    if car.isalpha():
        nb_lettres+=1
    if car.isdigit():
        nb_chiffres+=1
print("Dans la phrase %s, il y a %d lettres et %d chiffres" % (ph,nb_lettres , nb_chiffres))
```

- On pourrait utiliser `liste_caracters=list(ph)`.

☞ Dans la version ci-dessus, la présence de '.' n'est pas contrôlée lors de la lecture de la phrase. Même si on teste cette présence dans le code !

- Dans la version suivante, on va s'assurer de la lecture du '.' final tout en lisant la phrase sur plusieurs lignes.

```
stock=''
while True : # lire jusqu'au '.'
    ph=input("Donner une (suite de) phrase terminée par un point ")
    stock+=ph # stock contiendra toutes les lignes
    if ph[-1]== '.':
        break

nb_chiffres=nb_lettres=0
for car in stock:
    if car=='.':
        break
    if car.isalpha():
        nb_lettres+=1
    if car.isdigit():
        nb_chiffres+=1
print("Dans la phrase %s, il y a %d lettres et %d chiffres" % (stock,nb_lettres , nb_chiffres))
```

XIV-9 Exercice : compter les voyelles et consonnes

- On lit un texte terminé par EOF. Ce texte pouvant être composé de plusieurs phrases. Le programme doit déterminer le nombre de voyelles, le nombre de consonnes, le nombre de chiffres, le nombre d'autres caractères

☞ Remarques : on peut créer un dictionnaire Python (une MAP) indexé sur les voyelles init 0 :

```
nb_voyelles = {i:0 for i in 'aeiouAEIOU'}
```

Le résultat est `nb_voyelles={'a':0,'e':0,'i':0,'o':0,'u':0,'A':0,'E':0,'T':0,'O':0,'U':0}`

On peut alors compter le nombre d'occurrence de chaque voyelle dans la chaîne `ch1` :

```
ch1="Une nuit, un hotel a pris feu et les gens ont couru dehors en habits de nuit. Deux hommes observaient."
nb_voyelles = {i:0 for i in 'aeiouAEIOU'}
for lettre in ch1:
    if lettre in 'aeiouAEIOU': nb_voyelles[lettre] += 1
print(nb_voyelles)
{'O': 0, 'i': 5, 'E': 0, 'e': 13, 'A': 0, 'u': 7, 'a': 3, 'U': 1, 'I': 0, 'o': 6}
```

- Pour compter la somme de toutes les voyelles dans un string `ch2` (converti à la volée en minuscule) :

```
ch2 = "L'un a dit : puisque les gens ne font pas attention à leurs biens, je me suis servi en sortant !"
voyelles = "aeiou"; total_voyelles=0 # On ne prend que les voyelles minuscules
for v in voyelles: # On compte le nombre d'occurrence de chaque voyelle dans
    total_voyelles += ch2.lower().count(v) # ch2
# 37
```

→ Remarquer le ';' qui permet de placer plusieurs expressions sur la même ligne^a.

^a. La suite de l'histoire que les chaînes de caractères de cet exemple racontent :

`ch3="L'autre a répliqué : 'savez-vous que je suis de la Police ?' Le premier a répondu : 'savez-vous que j'invente des histoires ?' !"`

XIV-10 Exercices : lancers de dés

1. L'utilisateur donne un entier n entre 2 et 12, le programme donne le nombre de façons de faire n en lançant deux dés.
2. Même problème que le précédent mais avec n entre 3 et 18 et trois dés.
3. Généralisation des deux questions précédentes. L'utilisateur saisit deux entrées, d'une part le nombre de dés, **nbd** (limité à 10), et d'autre part la somme **s** comprise entre **nbd** et $6 \times \text{nbd}$.
Le programme calcule et affiche le nombre de façons de faire **s** avec les **nbd** dés.

XIV-11 Bonus : Tri sélection

- Dans cette méthode de tri d'un vecteur $V[0..k]$ éléments comparables (méthode de complexité $O(N^2)$), on trouve l'indice du minimum du $V[0..k]$ qu'on permute avec $V[0]$, puis on recommence avec $V[1..k]$... et on termine avec $V[k..k]$.
- Écrire la méthode.

☞ Remarques :

Noter `iMin = t.index(min(t[i:]))` : on veut l'indice du minimum de la liste t à partir du i ème élément.

XV Les exceptions

- Python bénéficie du mécanisme d'exception (venant des langages Modula et ADA). Le schéma simplifié d'un bloc avec exception est le suivant :

```
try :
    expression1
except :
    expression2
```

En cas d'exception dans `expression1`, on exécute `expression2`.

Si la clause `except` est absente pour un `try`, on passe au niveau supérieur (englobant) et ainsi de suite.

Ce mécanisme est important, en particulier dans le cas des fonctions partielles (cf. TDA).

Une exception peut être prédéfinie (voir ci-dessous) ou définie (à l'aide des classes) et levée par **raise**). Voir un exemple plus bas.

XV-1 Exemple 1

- Pour l'exercice pair-impair, on peut contrôler la lecture d'un entier au clavier.

```
a=input("Donner le coefficient a : ")
try:
    a=int(a)
    if (a%2 ==1):
        print(a, " est impair")
    else :
        print(a, " est pair")
except ValueError: # On vient ici si a=int(a) provoque une exception.
    print("pas un nbr")
```

Qu'est-ce passe-t-il si au lieu de donner un entier, on donne une lettre ?

Si la valeur rentrée au clavier n'est pas un entier, la conversion `a=int(a)` provoque une exception prédéfinie. De ce fait, la ligne suivante cette erreur n'est pas exécutée et Python cherche une clause **except ValueError** : qui va de paire avec **try** et exécute ce qui y est prévu.

- Et si on tient à lire un entier à tout prix :

```
# Tant qu'un entier n'est pas donné, insistez !
while True:
    a=input("Donner un entier : ")
    try:
        a=int(a)
        if (a%2 ==1):
            print(a, " est impair")
        else :
            print(a, " est pair")
        break
    except ValueError as e : # On vient ici si a=int(a) provoque une exception.
        print(e.args[0].split(": ")[1] + " n'pas un entier, recommencer !")

# Ici, on est sur que a contient un entier
```

☞ Remarques : ici, l'exception prend un nom locale (e) pour être traitée / détaillé.

- Dans cet exemple, l'exception e contient la chaîne "invalid literal for int() with base 10: 'a' ", l'expression e.args[0].split(": ") sépare cette chaîne de part et d'autre de ": " et donne une liste à 2 éléments contenant les deux chaînes ['invalid literal for int() with base 10', 'a'] et finalement l'indice [1] représente 'a' (le 2e élément).

XV-2 Exemple : Min/Max d'une séquence à la volée

- Dans cet exemple, on utilise l'exception EOFError pour détecter ctrl-D (ou ctrl-Z sous Windows) qui mettra fin aux lectures d'entiers.
- Cet exemple doit être exécuté sous Python et dans une fenêtre "terminal" *car spyder* se bloque à cause de l'exception EOFError provoquée par ctrl-D (ou ctrl-Z sous Windows).
- Cependant, sous *spyder*, on peut demander l'exécution dans le menu *Exécution/Configurer* une fenêtre "terminal" : Sous *spyder*, on peut associer à chaque fichier Python son type de console d'exécution (dans le menu *Exécution/Configurer*, choisir OK au lieu d'exécuter immédiatement). A partir de ce moment, l'exécution de ce code (placé dans un fichier .py) aura lieu sous la console précisée.

☞ Noter que si la fenêtre ferme immédiatement, c'est qu'il y a une erreur dans le code ! Il faut d'abord la corriger !

```
min=max=0
val=input("donner le premier entier ")
try :
    val=int(val)
    min=max=val

except ValueError : #On vient ici si on n'a pas donné un entier
    print("Fallait donner un entier")
    input("Taper sur entrée pour fermer la fenetre et quitter.")
    exit(0)

# Ici, on a lu un entier et initialisé min et max
while True:
    try :
        val=input("donner un entier ")
        val=int(val)
        if (val < min) :
            min=val
        if val > max :
            max=val

    except ValueError : #On vient ici si on n'a pas donné un entier
        print("Il faut donner des entiers")
        exit(0)

    except EOFError: #On vient ici si on a fait CTRL-D (ou CTRL-Z)
        print("c'est fini : min =",min, " max = ", max)
        break
    input("pour ne pas fermer la fenetre de suite ..")
```

☞ Rappel : pour pouvoir terminer les lectures avec ctrl-D (ctrl-Z), exécutez ce code dans une fenêtre "terminal".

- Si on veut continuer toutes les lectures après une erreur :

```
min=max=0

while True :
    try :
        val=int(input("donner le premier entier "))
        min=max=val
        break
    except ValueError :      #On vient ici si on n'a pas donné un entier
        continue

# Ici, on a lu un entier et initialisé min et max
while True:
    try :
        val=int(input("donner un entier "))
        if (val < min): min=val
        if val > max: max=val

    except ValueError :      #On vient ici si on n'a pas donné un entier
        print("Fallait donner des entiers, on recommence")
        continue

    except EOFError:         #On vient ici si on a fait CTRL-D (ou CTRL-Z)
        print("c'est fini : min =",min, " max = ", max)
        break

input("pour ne pas fermer la fenetre de suite ..")
```

XV-3 raise : deux exemples

- On utilise `raise` pour lever des exceptions, y compris celle que l'on juge opportunes.

Exemple 1 : si vous avez une fonction qui n'accepte qu'un entier $n > 0$ (par exemple, n est un nombre d'itérations), procédez comme suit :

```
def ma_fonction(n) :
    if n < 1 : raise ValueError
    else : # la suite .....
```

→ l'exception `ValueError` sera levée si on appelle cette fonction avec $n < 1$.

L'utilisateur de `ma_fonction()` prendra ses précautions.

Exemple 2 : Supposons que l'on veuille trouver le plus grand réel (à 1% près) de Python. L'exemple suivant multiplie un nombre (initialisé à 1000000) par un facteur de 1,01 jusqu'à ce que cette multiplication ne soit plus possible faute de capacité ! Dans ce dernier cas, le nombre devient **infini** (`inf` de Python) et on lève, par `raise`, une exception prédéfinie `ValueError` dont le traitement affiche la plus grande valeur de facteur atteinte avant de devenir `inf = float('inf')`.

```
facteur = 1000000
while True :
    try :
        if 1/facteur :
            save = facteur
            facteur *= 1.01      # l'opération qui provoquera une exception
            print('nombre grand actuel :', save)
        else :
            raise ValueError      # on lève une exception prédéfinie

    except ValueError :
        print('limite :', save)
        break

print("Passer à la suite ....")

""" TRACE des derniers nombres affichés :
.....
nombre grand actuel : 1.7600504054160286e+308
nombre grand actuel : 1.7776509094701888e+308
nombre grand actuel : 1.7954274185648906e+308
limite : 1.7954274185648906e+308      # le plus grand réel (à 1% près)

Passer à la suite ....
```

XV-4 Exercice : moyenne et fonction

- On reprend un exemple vu plus haut.
- Lire et constituer une liste d'entiers puis calculer la moyenne et la variance de ces valeurs.
- Protéger la lecture des entiers par une exception.

XV-5 Expression Pass

- `pass` ne fait rien. C'est donc un *bouche-trou* à placer là où il faut quelque chose. On peut l'utiliser (entre autres) dans les exceptions.

```
try :
    vaZy_Plante()

except :
    pass
```

- Ici, l'exception est captée et ignorée.
- D'autres exemples de `pass` :

```
while True:
    pass # Attente active, par exemple pour un Ctrl+C

def fonction_a_ecrire(params):
    pass # La fonction existe. Pensez à la compléter.
```

- `pass` est également utilisé pour créer des classes vides qui doivent exister ! Voir Inf-TC2. On l'utilise aussi dans les exceptions si on veut les ignorer.

XVI Génération de nombres aléatoires

- Des exemples (tirage uniforme) :

```
import random;
random.random() # Random float x, 0.0 <= x < 1.0
#Donne -> 0.37444887175646646

random.uniform(1, 10) # Random float x, 1.0 <= x < 10.0
#Donne -> 1.1800146073117523

random.randint(1, 10) # Entiers de 1 à 10 (bornes incluses)
#Donne -> 7

random.randrange(0, 101, 2) # Entiers pairs de 0 à 100
#Donne -> 26

random.choice('abcdefghij') # Choix aléatoire
#Donne -> 'c'

items = [1, 2, 3, 4, 5, 6, 7]
random.shuffle(items)
items
#Donne -> [7, 3, 2, 5, 6, 4, 1]

random.sample([1, 2, 3, 4, 5], 3) # échantillonner 3 éléments
#Donne -> [4, 1, 5]
```

- Une manière simple de générer une liste d'entiers aléatoires :

```
a, b, n = 3, 20, 5
[randint(a, b) for i in range(n)]

# Donne par exemple : [16, 9, 16, 14, 8]
```

☞ Remarques : noter la définition `a, b, n = 3, 20, 5`.

- `[randint(a, b) for i in range(n)]` est appelé liste en compréhension à voir en section [XXIII](#) page [47](#).

XVI-1 Exemple : indice du minimum d'une liste aléatoire d'entiers

```
from random import seed, randint

# fonctions
def listAleatoireInt(n, a, b):
    """Retourne une liste de <n> entiers aleatoires dans [<a> .. <b>]."""
    return [randint(a, b) for i in range(n)]

# programme principal
seed() # initialise le generateur de nombres aleatoires
t = listAleatoireInt(100, 2, 125) # construction de la liste

## calcul de l'indice du minimum de la liste
iMin = t.index(min(t))

## Affichages
print("{} t[iMin] = {}".format("Indice :", t[iMin])) # Voir plus loin pour la fonction format
```

Remarques :

Noter les deux {} : la première est pour "Indice :" et la seconde pour t[iMin].

→ voir section [XIII](#) page 20

XVI-2 Exemple : amplitude et la moyenne d'une liste aléatoire de réels

- On échantillonne une liste de réels dont on affiche l'amplitude et la moyenne.

```
# imports
from random import seed, random

# fonctions
def listAleatoireFloat(n):
    """Retourne une liste de <n> flottants aleatoires."""
    return [random() for i in range(n)]

# programme principal
n = int(input("Entrez un entier [2 .. 100] : "))
while not (2 <= n <= 100): # saisie filteree
    n = int(input("Entrez un entier [2 .. 100], s.v.p. : "))

seed() # initialise le generateur de nombres aleatoires
t = listAleatoireFloat(n) # construction de la liste

print("Liste :", t)
print("Amplitude : {:.2f}".format(max(t) - min(t)))
print("Moyenne : {:.2f}".format(sum(t)/n))
```

- Pour le formatage des écritures, voir la section [XIII](#) page 20.

XVI-3 Échantillonnage : Normale et Uniforme

Remarques :

- La fonction `seed([n])` (avec un paramètre optionnel `n`) permet d'initialiser une séquence aléatoire. Si le paramètre `n` n'est pas fourni (ou si on fournit `None`), Python initialise la séquence avec le temps système. Ce même temps système est utilisé à la première importation du module *random*.

- Pour générer une séquence de distribution *Normale*, utiliser

```
from numpy.random import normal
N = normal(0, 1)          # centré réduit  $\mathcal{N}(0, 1)$ 
N = normal()              # la même chose (paramètre par défaut :  $\mathcal{N}(0, 1)$ )
N                          # Donne p.ex -0.11692213582578123
N = normal(10, 5)         #  $\mathcal{N}(10, 5)$ 
normal(size= (2, 2))      # renvoie une matrice  $2 \times 2$  de  $\mathcal{N}(0, 1)$ 
...
```

A propos de `randn()` du module *numpy (scipy)*

```
from scipy import randn
N = randn()                # alternative à N = normal(0, 1)
N = randn(3, 2)            # renvoie une matrice  $3 \times 2$  dont les nombres viennent de  $\mathcal{N}(0, 1)$ 
N = randn(2, 2, 2)         # renvoie une cube  $2 \times 2 \times 2$  dont les nombres viennent de  $\mathcal{N}(0, 1)$ 
```

Donc :

$\sigma * \text{randn}(d_1, \dots, d_n) + \mu$ est équivalent à $\text{normal}(\mu, \sigma, d_1, \dots, d_n)$ d_i : taille de la i ème dimension

- Échantillonnage Uniforme : `random.randint(a, b)` renvoie un entier $a \leq N \leq b$

- A noter :

```
import random
o random.gauss( $\mu, \sigma$ ) renvoie un réel de  $\mathcal{N}(\mu, \sigma)$ 
o random.random() renvoie un réel dans  $[0 .. 1[$ .
o random.choice(seq) renvoie un élément aléatoire de la séquence seq
o random.randrange(0, 101, 2) est équivalent à choice(range(0, 101, 2)).
o random.shuffle(seq[, random]) "mélange" la séquence X sur place et seq devient égale à l'une de ses propres permutations.
```

Le 2e argument est par défaut la fonction `random()` qui renvoie un réel `N` dans $[0 .. 1[$.

Si le 2e argument est présent, il doit être le nom d'une fonction sans argument.

Par exemple, si on définit

```
def f() : return 0 et L=[7, 2, 10, 7, 7, 18, 7, 7, 15, 11], alors
random.shuffle(L, f) donne dans L la permutation [2, 10, 7, 7, 18, 7, 7, 15, 11, 7].
```

Puis à la répétition de ce même appel, le 1er élément de L est placé à la fin le 2e devient le premier, ...

```
o random.sample(seq, k) renvoie une liste de taille k des éléments sans remise de seq.
```

Consulter la doc du module *random* pour d'autres distributions.

XVI-4 Complément : Tirage avec remise

- Exemple de tirage aléatoire avec remise (celui de Python est biaisé) et il est fortement préconisé d'utiliser les générateurs de *numpy* dans une application sensible à ce type de tirage.

- Pour une distribution Uniforme :

```

from math import *
from random import *

# Simulation d'un tirage avec remise de k objets parmi n

def tirage_uniforme_k_of_n_avec_remise(n,k):
    if k > n :
        raise ValueError("k (%d) doit être =< que n (%d) " % (k,n))
    T=[x for x in range(1,n+1,1)] # On constitue une urne avec p objets

    for i in range(k) : # Tirage de k numéros
        d=randint(1,n)
        # On permute le d-ème élément de l'urne et la dernière
        temp=T[d-1]
        T[d-1]=T[n-1]
        T[n-1]=temp
        n=n-1
    return T[n:n+k+1]

n=int(input("La valeur de n ? "))
k=int(input("La valeur de k (parmi n) ? "))
try :
    print("Le tirage : ", tirage_uniforme_k_of_n_avec_remise(n,k))
except ValueError as err :
    print("Error: {0}".format(err))

```

- Pour une distribution Normale :

```

from math import *
from random import *

def tirage_gauss_centre_reduit():
    x=uniform(0,1)
    y=sqrt(-2*log(uniform(0,1)))
    y = y*cos(x*2*pi)
    return y

def tirage_centre_reduit_k_of_n_avec_remise(n,k):
    if k > n :
        raise ValueError("k (%d) doit être =< que n (%d) " % (k,n))

    T=[x for x in range(1,n+1,1)] # On constitue une urne avec p objets
    for i in range(k) : # Tirage de k numéros
        d=randint(1,n)
        # On permute le d-ème élément de l'urne et la dernière
        temp=T[d-1]
        T[d-1]=T[n-1]
        T[n-1]=temp
        n=n-1
    return T[n:n+k+1]

```

XVI-5 Exemple : Tirage de cartes

- A partir d'un jeu de 52 cartes, donner la probabilité d'avoir un as de pique (10000 essais).
- Dans cet exemple, on utilise la fonction `choice()` qui choisit un élément d'un range aléatoire dans une séquence.
- La variable `toutes_cartes` représente tous les 52 cartes sous forme de strings, à l'aide d'une *liste en compréhension* (lire ci-après).

```

valeurs=[1,2,3,4,5,6,7,8,9,10, 'Valet', 'Dame', 'Roi']
couleurs={'carreau', 'coeur', 'pique', 'trefle'}
toutes_cartes=[str(v)+' '+c for v in valeurs for c in couleurs]

from random import *

somme=0
for n in range(1000):
    if choice(toutes_cartes)=="1 pique":
        somme+=1

print("proba (fréquence) obtenue : ",somme/10000)
print("proba d'une carte parmi 52 : ",1/52)

# —> 0.002
# 0.019230769230769232

```

- Noter que la liste en compréhension utilisée équivaut tout simplement à :

```

toutes_cartes=[]
for v in valeurs :
    for c in couleurs :
        toutes_cartes.append(str(v)+' '+c)

```

→ Voir la section [XXIII](#) page [47](#) pour plus de détails sur les *listes en compréhension*.

XVI-6 Exercice : Tirage de cartes

- Donner la probabilité d'avoir au moins 3 cartes de la même couleur dans une main de 4 cartes (10000 essais)

XVI-7 Bonus : Tirage de cartes (bis)

- Donner la probabilité d'avoir au moins 3 As dans une main de 5 cartes (10000 essais).

XVI-8 Bonus : Fréquences de dés

- Lancer 10000 fois un dé et donner la fréquence de chaque face.

XVII Quelques fonctions sur les chaînes

- Une bonne partie de ces fonctions sont celles applicables aux listes.
 - **str.center(n)** : centrage dans un champ de taille *n*, voir plus haut.
 - **str.ljust(n)** : justification, voir plus haut.
 - **str.rjust(n)** : justification, voir plus haut.
 - **str.count(ele)** : compte le nombre d'occurrences de *ele* dans *str*.
 - **str.lower()** : transformation en minuscules
 - **str.upper()** : transformation en majuscules
 - **str.find(ele)** : renvoie l'indice de la première occurrence de *ele* dans *str*.
 - **str.split(caractère)** : découpage de *str* suivant caractère

- Exemples :

```
my_name
# 'David'
my_name.upper()
# 'DAVID'
my_name.center(10)
# ' David '
my_name.find('v')
# 2
my_name.split('v')
# ['Da', 'id']
```

📖 **Rappel** : une constante string est un objet non mutable :

```
my_name[0]='d'
# TypeError: 'str' object does not support item assignment
```

📖 **Remarque** : comme pour les listes, les tuples, on peut multiplier un string par un entier *n* pour répéter son contenu *n* fois :

```
my_name
# 'David'

my_name*3
# 'DavidDavidDavid'
```

📖 **Rappel** : Par contre, parmi les listes, tuples et constantes strings, seules les listes sont mutables (on peut modifier la valeur d'un de ses éléments).

XVIII Imbrication de fonctions

- Sous Python, la définition d'une fonction peut contenir la définition d'autres fonctions.

```
def f1(x) :
    x=1
    def g1(x) :
        x += 10
    g1(x)
    print(x)

# — main() —
f1(2)      # donne 1
g1(2)      # donne une erreur : g1() est inconnue
```

- Remarquez que la fonction `g1()` n'est visible qu'à l'intérieur de la définition de la fonction `f1()`. On ne peut donc pas l'appeler à partir du niveau 0.
- Noter que dans `def g1(x) :`, le paramètre formel `x` devient une variable locale à `g1()` ; elle reçoit sa valeur initiale (1) par copie et n'aura plus de lien avec `x` de `f1()`. Voir section XIX, page 39 sur la portée des variables.

XVIII-1 Quelques règles utiles

- Les règles (sur la visibilité des variables) sont développées en section XIX, page 39.
 - Dans une fonction Python : toute variable définie (par exemple par `x=1`) et tout paramètre (initialisé par le passage de paramètre par valeur) est **locale** à la fonction.

```
def fonc(x) :
    x += 15    # x est locale 'fonc'

x=12
fonc(x);x
#12          —> x n'a pas changé de valeur
```

- Les variables du *main* (niveau global) sont accessibles dans une fonction via le mot clé *global*. Si elles existent au préalable et au niveau global, elles sont ainsi référencées. Si elles n'existent pas, elles sont créées au niveau global. On peut ainsi créer une variable (globale) pour le compte du niveau global dans une fonction.

```
def machin() :
    global bidule
    bidule = 50

machin(); bidule
# 50
```

→ Il y eu donc une **création**. Par contre, s'il s'agit d'une modification, ça ne passera pas :

```
def truc() :
    global muche
    muche += 50

truc()
# NameError: name 'muche' is not defined
```

→ Cette fois, on a essayé de **modifier** `muche`. Dans ce cas, `muche` doit avoir existé avant l'appel de `truc()` puisqu'on ajoute 50 à son ancienne valeur :

```
def truc() :
    global muche
    muche += 50

muche=10
truc(); muche
# 60
```

- Supposons qu'une fonction définisse d'autres fonctions (par exemple, `f()` définit `g()` qui définit `u()`). Pour accéder à la variable `x` de `f()`, les fonctions `g()` et `u()` utilisent le mot clé *nonlocal* `x`. La variable `x` doit exister dans `f()` et y avoir déjà reçu une valeur. De même, pour accéder à la variable `z` de `g()`, `u()` utilisera *nonlocal* `z`.

```
def f() :
    x = 1
    def g() :
        nonlocal x
        x+=10
        print('x dans g() : ',x)
        def u() :
            nonlocal x
            x+=100
            print('x dans u() : ', x)
            # On est dans g()
            u()
        # On est dans f()
        g()
        x += 1000
        print('x dans f() : ', x)
    f()
# x dans g() : 11
# x dans u() : 111
# x dans f() : 1111
```

- Les 3 fonctions écriront `global k` pour utiliser une variable `k` du niveau 0 si `k` est définie avant l'appel de `f()`. Si `k` n'existe pas au niveau global avant cet appel, elle existera désormais (voir le premier exemple ci-dessous).
 - Le mot clé `nonlocal` exprime le besoin d'utiliser d'autres variables que les locales. Ce mot clés ne définit pas de variable et ne change pas le statut d'une variable. Il fait seulement référence à une variable existante des niveaux englobants.
 - Un paramètre formel d'une fonction est une variable locale à la fonction, il peut donc être référencé par `nonlocal`.
- Un premier exemple : **global**

```
def f():
    def g() :
        global val_g      # "val_g" n'existe pas encore. La voici créée.
        val_g=300
    g()
    global val_g           # "val_g" existe par l'appel à g(). Elle est référencée.
    val_g += 100
    val_l = 200            # "val_l" est ici local. On ne verra pas son effet à l'extérieur.

val_l = 0
f()
val_l
val_g

# On obtient
# 0
# 400
```

→ Il est important qu'ici, `global val_g` soit citée au moins 2 fois ! Faute de quoi on n'aura pas affaire à une variable globale.

- Un exemple qui mélange `nonlocal` et `global` :

```
def f(x,y) :
    x=1
    y=2
    def g(x) :
        x += 10          # x est maintenant locale à g()
        print('x dans g() : ', x)  # 11 : on a reçu 1 puis +10
        nonlocal y       # y doit exister dans la fonction englobante
        y += 20          # Ca fait +20 sur le y de f()
        print('le nonlocal y dans g() : ', y)
        global k          # K devient globale
        k+=30            # Ca fait +30 sur le k de main
        print('le global k dans g() : ', k)
    g(x)
    print('retour de g() : x dans f() : ', x, ' y dans f() : ', y)
    global z
    z += 50
    print('le global z dans f() : ', z)

# main() :
z = 3
k = 4
f(z,k)
print('retour de f() : z dans main() : ', z, ' k dans main() : ', k)

""" donne
x dans g() : 11
le nonlocal y dans g() : 22
le global k dans g() : 34
retour de g() : x dans f() : 1 y dans f() : 22
le global z dans f() : 53
"""
```

- Une dernière chose : sous Python, on a des **paramètres nommés**. Ce mécanisme permet une certaine souplesse lors des appels dans la mesure où en nommant les arguments, on n'est pas obligé de respecter l'ordre des paramètres formels donné dans la signature de la fonction.

Dans l'exemple suivant, remarquer l'ordre des arguments d'appel et celui des paramètres formels :

```
def f(petit, moyen, grand) :
    print(petit, ' <= ', moyen, ' <= ', grand)

f(grand=12, petit= 5, moyen = 9)

# 5 <= 9 <= 12
```

XVIII-2 Bonus : Tri par insertion

- L'idée de cette méthode de tri s'explique plus facilement de manière récursive, quitte à écrire ensuite une solution itérative.
- Pour trier une séquence $T[0..n]$: supposons que $T[1..n]$ est trié. Il ne nous reste qu'à insérer $T[0]$ à sa place dans $T[0..n]$. Appelons cette fonction `insere_a_sa_place(liste, val)`.
- Et comment faire pour trier $T[1..n]$? Il faut trier $T[2..n]$ puis insérer $T[1]$ à sa place dans $T[1..n]$.
Noter bien que pour insérer $T[1]$ à sa place dans $T[1..n]$, on sait que $T[2..n]$ est déjà trié et il faut 'injecter' $T[1]$ à sa place dans $T[1..n]$, en poussant des éléments si nécessaire. C'est le rôle de `insere_a_sa_place(liste, val)`.
- Cet enchaînement nous conduit à un état de base (d'induction) : on finira par arriver à l'état où pour trier une tranche du tableau à 2 éléments $T[n-1..n]$, il faudra trier $T[n]$ (il est déjà trié car c'est un singleton). Donc, la première élément à être inséré à sa place sera $T[n-1]$ à insérer dans $T[n-1..n]$ par `insere_a_sa_place(liste, val)`.
- Ainsi, $T[n-1..n]$ devient trié et l'insertion suivante va insérer $T[n-2]$ à sa place dans $T[n-2..n]$ et enfin, on finira par insérer $T[0]$ à sa place dans $T[0..n]$.
- Un exemple : soit $T = [5, 2, 3, 9, 1, 6]$. On doit
 - trier $T = [2, 3, 9, 1, 6]$ (qui doit donner $T[1, 2, 3, 6, 9]$) puis y insérer 5 à sa place.
 - trier $T = [3, 9, 1, 6]$ (qui doit donner $T[1, 3, 6, 9]$) puis y insérer 2 à sa place.
 -
 - trier $T = [6]$ (il l'est déjà) puis y insérer 1 à sa place.
 - ← insérer 1 dans $T = [6]$ donne $[1, 6]$
 - ← ...
 - ← insérer 2 dans $T[1, 3, 6, 9]$ donne $T[1, 2, 3, 6, 9]$
 - ← insérer 5 dans $T[1, 2, 3, 6, 9]$ donne $T[1, 2, 3, 5, 6, 9]$
- Une autre illustration de cette méthode sur le string "INSERTIONSORT" :

```

I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T I O N S O R T
E I I N O R S T I N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T
E I I N N O O R R S S T T
E I I N N O O R R S S T T
```

- Écrire la méthode de Tri par insertion. Utiliser une fonction imbriquée pour l'insertion des éléments à leur place dans la partie triée. Quelle est la complexité de cette méthode de tri ?

XVIII-3 Bonus : Tri split-merge

- Tri d'une liste par la méthode *split-merge* (proche de *merge-sort*) :
 - Pour une liste L à trier, on cherche dans L une sous-séquence ordonnée et on la met dans la sous liste L_1
 - Ensuite, la sous-séquence ordonnée suivante de L est mis dans L_2 , ...
 - Et ainsi de suite alternativement, une séquence dans L_1 , la suivante dans L_2 jusqu'à l'épuisement de L .
- Exemple : $L = [4, 5, 2, 3, 1, 9, 6, 0, 7]$, la fonction *my_split* donne (*split* correspond à une fonction prédéfinie de Python) :
 - L_1 : 4,5 puis 1,9 puis 0,7 $\rightarrow L_1 = [4, 5, 1, 9, 0, 7]$
 - L_2 : 2,3 puis 6 , $\rightarrow L_2 = [2, 3, 6]$
- Ensuite, on fusionne L_1 et L_2 (un peu comme merge-sort en mettant toujours le plus petit élément d'abord) :

La fusion donne : L : 2,3 (de L_2 car 2 et 3 sont plus petits que 4) puis 4,5,1 (de L_1 car ils sont plus petits que 6),6,9,0,7. Donc, le résultat de cette fusion est $L = [2, 3, 4, 5, 1, 6, 9, 0, 7]$.
- Mais la liste L résultante de cette fusion n'est pas encore triée. On recommence avec *my_split* sur L :
 - L_1 : [2,3,4,5 puis 0,7]
 - L_2 : [1,6,9]

\rightarrow On fusionne : $L = [1, 2, 3, 4, 5, 0, 6, 7, 9]$
- On re scinde L :
 - L_1 : [1,2,3,4,5]
 - L_2 : [0,6,7,9]

\rightarrow On re fusionne : $L = [0, 1, 2, 3, 4, 5, 6, 7, 9]$
- Un dernier appel à *my_split* produit $L_1 = L$ et $L_2 = []$

\rightarrow terminé.
- **La condition d'arrêt** de cet algorithme est que la liste L_2 renvoyée par *my_split* soit vide.

XIX Complément sur la Visibilité des variables

- On étudie plus en détails la question de la portée (visibilité, champ, contexte) des variables.
- En matière de fonctions Python, on peut se poser la question de la portée des variables.
- A l'intérieur d'une fonction, toute variable est local à la fonction.

```
def fonc(x) :
    x = x+1
    print(x)

x=15
fonc(x)
print(x)

# Donne
16      # écrit pas fonc
15      # au retour de fonc, x n'a pas changé de valeur
```

XIX-1 Variables locales

- Observe les exemples ci-dessous :

```
a=1
def f() :
    print(a)

f()
# Donne 1
```

→ Dans cet exemple, `print(a)` affichera la valeur de la variable `a`.

```
def g() :
    x=1
    def ff() :
        print(x)
    ff()
g()
# Donne 1
```

→ Ici aussi, `x` de `g()` est visible dans `ff()`.

⚠ Par contre :

```
a=1
def f() :
    a += 1

f()
# Erreur
```

→ Dans cet exemple, `a += 1` provoque une **erreur** d'accès à `a` : la variable locale `a` référencée avant de recevoir une valeur. Ce qui veut dire qu'il faut d'abord écrire `a = . .` et à partir de ce moment, les deux `a` seront différentes.

```
def g() :
    x=1
    def ff() :
        x += 1
    ff()
g()
# Erreur
```

→ Ici aussi, `x += 1` provoque une erreur d'accès à `x`.

⚠ Règle : une fonction a accès en lecture aux variables définies dans son contexte ainsi que dans le contexte englobant (cf : `print(a)` et `print(x)` ci-dessus).

→ Si la variable accédée est **non mutable** (int, constante string, tuple, etc), toute modification de celle-ci provoque une erreur.

→ Par contre, pour une variable **mutable** (*List, Dict, Set, etc.*), la modification est possible.

◦ Exemple :

```
L=[]
def f() :
    L.append(1)

f()
print(L)

# Donne [1]
```


→ La liste étant mutable, *L* peut être modifiée.

- Les mots clés *global* et *nonlocal* permettent de modifier ce comportement.

XIX-2 global

- Un exemple :

```
def method():
    # On change le statut de "value" : elle sera globale.
    # Sans "global", l'affectation sera "local"
    global value
    value = 100

value = 0
method()
value
# Donne 100
```

- Ici, globale *value* fait référence à la variable *value* du niveau global.
- Mais observez cet exemple :

```
def method():
    # On change le statut de "value1" : elle sera globale.
    # Sans "global", l'affectation sera "local"
    global value1
    value1 = 100
    value = 200

value = 0
method()
value
value1
# Donne
# 0
# 100
```

- Ici, global *value1* (*value1* n'existe pas encore) définit et affecte une valeur à *value1*.

☞ Remarques : global *var* définit *var* si elle n'existe pas et la rend accessible à l'extérieur de toute fonction. Si *var* existe déjà (à un niveau global), ce sera cette *var* qui sera référencée.

XIX-3 nonlocal

- *nonlocal* a un comportement similaire à *global* mais pour des fonctions imbriquées. *nonlocal* veut dire ni global, ni local. Une variable *nonlocal* fait référence à un niveau englobant.

```
def method():
    def method2():
        # dans cette fonction imbriquée, 'value' fait reference à une variable 'nonlocal'.
        nonlocal value
        value = 100

    # Set local.
    value = 10
    method2()

    # La variable local est affectée par un changement NON LOCAL.
    print(value)

# essai
method()

# Donne 100
```

- *nonlocal* rend pratique l'échange de variables entre les fonctions imbriquées. Sans *nonlocal*, on risque de provoquer des conflits de noms entre un fonction imbriquée et le niveau englobant là où *global* ne rentre pas dans ce niveau de détail.
- Il est communément admis que la plupart des fonctions Python n'ont besoin de ce mécanisme qui rend le code Python obtenu plus complexe. Néanmoins, *global* et *nonlocal* peuvent convenir dans certaines situations pour éviter les conflits de noms.

XIX-4 Un exemple récapitulatif

- A l'intérieur d'une fonction, l'expression `global x` précise que `x` existe (sinon, existera) globalement et en dehors de la fonction et doit être réaffectée en dehors de son champ (cf. `spam` dans les commentaires `# (glob)` de l'exemple).
- Une variable `nonlocal` indique que cette variable existera dans un champ interne (englobé) et doit être réaffectée dans ce contexte là (cf. `spam`).
- Un exemple récapitulatif :

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        print("Dans do_nonlocal() : ", spam)
        spam = "nonlocal spam"
    def do_global():
        global spam          # (glob)      se superpose à la définition 'spam = "test spam"'
        # print("Dans do_global()", spam)  Provoquera une erreur : spam non défini.
        spam = "global spam" # (glob)      le 'spam' du niveau 0, pas celle de scope_test.

    spam = "test spam"       # (glob),    nécessaire pour 'global spam' ci-dessus
    do_local()
    print("Après l'affectation local dans do_local() : ", spam)
    do_nonlocal()
    print("Après affectation nonlocal dans do_nonlocal() : ", spam)
    do_global()
    print("Après affectation globale dans do_global() : ", spam)

scope_test()
print("Dans la portée globale (hors les fonctions) : ", spam)

""" Donne
Après l'affectation local dans do_local() : test spam
Dans do_nonlocal() : test spam
Après affectation nonlocal dans do_nonlocal() : nonlocal spam
Après affectation globale dans do_global() : nonlocal spam
Dans la portée globale (hors les fonctions) : global spam
"""
```

- On note que :
 - `do_local` n'a pas changé la valeur de `spam` de `scope_test`. Ce que `do_local` a fait à `spam` n'est pas visible à l'extérieur. Si on tente d'écrire `spam` au début de `do_local`, on provoque une erreur de définition.
 - Dans `do_nonlocal`, la référence à `spam` concerne `spam` de `scope_test`. L'affectation fait dans `do_nonlocal` modifie donc `spam` de `scope_test`.
 - Dans `do_global`, on déclare `global spam` (une variable globale va exister !). Une tentative d'écriture (par `print`) de ce `spam` provoque une erreur : la nouvelle occurrence de `spam` n'a pas encore de valeur ! Noter bien qu'au retour de l'appel à `do_global`, `spam` vaut toujours `nonlocal spam`. Noter également que `global spam` fera exister `spam` au niveau global (et affichée à la fin de la partie principale).
 - La fonction `scope_test` écrit ensuite la valeur de `spam` définie dans `do_global`.
- ☞ Si on écrivait (sous Python) la valeur de `spam`, on obtiendrait `global spam`.

XX Introduction à la récursivité

- Récursivité et Induction Mathématique (et lien directe avec la preuve)
- Récursivité *directe* et *indirecte*.

XX-1 Exemple : Tri bulle récursive

- Pour trier une liste (ou tableau) L , le principe de cette méthode est le suivant :
 - $i=0$
 - visiter les éléments de $L[i+1:]$ et si l'un est $< L[i]$ alors permuter $L[i]$ et cet élément
 → à la fin de cette action, $L[i]$ contient le minimum de L
 - recommencer en faisant $+1$ sur i (jusqu'à $i == \text{len}(L)-1$)
- Soit la version itérative de l'algorithme du Tri Bulle :

```
def tri_bulle_iteratif(lst) :
    taille = len(lst)
    for i in range(taille-1) :
        for j in range(i+1, taille) :
            if lst[i] > lst[j] :
                lst[i], lst[j] = lst[j], lst[i] # permutation
    return lst

liste=[1,3,13,4,7,2,9,2,18]
print(tri_bulle_iteratif(liste))
# [1, 2, 2, 3, 4, 7, 9, 13, 18]
```

- On reprend la méthode tri bulle pour donner sa version récursive.
- Récursivité *directe* : la fonction `tri_bulle()` s'appelle (dans le texte).

```
from random import seed, randint

# procédure récursive (qui modifie son argument L et ne renvoie rien)
def tri_bulle(L,from_) :
    if from_ < len(L)-1 :
        iMin = t.index(min(L[from_:]))
        t[from_], t[iMin] = t[iMin], t[from_]
        tri_bulle(L,from_+1)

# programme principal
seed() # initialise le generateur de nombres aleatoires
N=10
t = [randint(2, 125) for i in range(N)]
print("Avant le tri, liste = {}".format(t))
tri_bulle(t,0)

print("Après le tri, liste = {}".format(t))

# Avant le tri, liste = [125, 78, 77, 24, 2, 22, 86, 120, 111, 84]
# Après le tri, liste = [2, 22, 24, 26, 77, 78, 84, 111, 120, 125]
```

XX-2 Un exemple de récursivité indirecte

- Calcul simple de *sinus* et *cosinus*.
- Dans cette forme de la récursivité, la fonction `my_sin(x)` appelle `my_cos(x)` qui à son tour appelle `my_sin(x)`.
 Noter que cet exemple contient également la récursivité directe.

```
import math;
def my_sin(x) :
    if abs(x) < 1e-6 : sine=x
    else : sine = 2 * my_sin(x/2) * my_cos(x/2)
    return sine

def my_cos(x) :
    if abs(x) < 1e-6 : cosine=1
    else : cosine = pow(my_cos(x/2), 2) - pow(my_sin(x/2), 2)
    return cosine

angle = math.pi/3
print("avec notre méthode : {0:8.6f}".format(my_sin(angle)+my_cos(angle)))
print("avec la méthode classique : {0:8.6f}".format(math.sin(angle)+math.cos(angle)))

""" TRACE
avec notre méthode : 1.366026
avec la méthode classique: 1.366025
"""
```

XX-3 Exercice : recherche Dichotomique dans une liste triée

- Méthode : pour chercher *val* dans une liste triée de taille N $L[0:N]$,
 - On examine le milieu de la liste : $L[N//2]$
 - Si $val == L[N//2]$ alors on a trouvé ; renvoyer vrai
 - Si $val > L[N//2]$ alors il faut chercher *val* à droite dans la tranche $L[N//2+1:]$
 - Si $val < L[N//2]$ alors il faut chercher *val* à gauche dans la tranche $L[:N//2]$
 On constate que la taille de la liste est divisée par 2 à chaque appel récursif.
 - Continuer ainsi soit jusqu'à $L == []$: renvoyer faux
- Écrire la recherche Dichotomique d'un élément dans une liste.

XX-4 Bonus : Tri par insertion


- Pour réaliser cet exercice en bonus, vous devez avoir réalisé le tri par insertion itératif vu (en bonus) plus haut.
- La version itérative de l'algorithme de tri par insertion a été vue plus haut. Écrire la version récursive de cette même méthode.

XX-5 Bonus : Tri par insertion utilisant la recherche dichotomique

- L'algorithme itératif du tri par insertion précédent est $O(N^2)$ car la fonction `insérer_a_sa_place()` est $O(N)$ et celle-ci est appelée N fois.
- Reprendre l'algorithme de recherche `dichotomie()` ci-dessus (qui est $O(\log(N))$) et modifiez le pour qu'il renvoie la place d'insertion d'un élément dans un tableau trié. Puis utiliser cet algorithme dans le tri par insertion.
 - Pour l'ajout concret de l'élément à insérer, on pourra utiliser `L.insert(Place, val)`.
- Quelle est la complexité de ce nouvel algorithme ?

XX-6 Bonus : lancers de dés

- L'utilisateur donne deux valeurs : le nombre de dés *nbd* (limité à 8), et la somme *s* à réaliser avec les dés (*s* comprise entre *nbd* et $6*nbd$). Le programme calcule récursivement et affiche le nombre de façons de faire la somme *s* avec les *nbd* dés.
- La solution suivante utilise un argument *niveau* (*niv*) qui permet de faire un affichage décalé de $niv * ' '$ pour faciliter la lecture de la trace.
 - Les traces débutant `cas-3: . . .` cumulent les sous calculs du dessus et qui sont décalés plus à droite.

 **Remarques** : il est important d'avoir à l'esprit que Python limite le nombre d'appels récursifs à une fonction à 1000 fois. C'est une limitation de l'implantation du langage.

Par ailleurs, Python ne traite et ne supprime pas la récursivité terminale (pas encore !).

Un appel à `gc` pourrait éventuellement améliorer l'état de la mémoire.

XXI Paramètre formel à valeur par défaut des fonctions

- Si vous demandez un café dans un bistrot en France, on vous apporte un café noir (court, ni allongé ni double) avec sucre et sans lait (et on ne vous apporte pas de lait). Si vous souhaitez un café décaféiné, ou avec du lait, ..., il faudra demander explicitement (et payer pour le lait!).
- Par contre, en Allemagne (par exemple), on vous apporte du lait systématiquement.
- On pourrait alors proposer une fonction :

```
def servir_cafe(deca=False, taille= 'court', lait=False) :
    # couleur : 'noir' ou déca
    # taille : double, allongé, ...
    print('On sert un café ', end= ' ')
    print('décaféiné', end= ' ') if deca else print('noir', end= ' ')
    if taille != 'court' : print(taille, end= ' ')
    if lait : print(' avec du lait')
    else: print()

servir_cafe()
# On sert un café noir

servir_cafe(taille='double')
# On sert un café noir double

servir_cafe(True, 'allongé')      # sans utiliser les noms : respecter l'ordre des paramètres.
# On sert un café décaféiné allongé
```

- Notez qu'on ne peut pas écrire :

```
servir_cafe(True, lait=True, 'allongé')
```

sous peine de provoquer une erreur : `SyntaxError: non-keyword arg after keyword arg`

De même, `servir_cafe(deca=True, 'allongé')` provoque la même erreur.

- On ne peut donc pas revenir aux positions (de gauche à droite sans les nommer) si on a commencé par les nommer. Mais on peut appeler la fonction en nommant les paramètres après avoir utilisé les positions :

```
servir_cafe(True, 'allongé', lait=True)      qui donne
On sert un café décaféiné allongé avec du lait
```

- Sous Python, les paramètres (formels) à valeur par défaut sont nommés. Ces noms permettent de passer les arguments dans un ordre quelconque (on a vu un exemple plus haut).
- En Python, la combinaison des valeur par défaut et le nom pour un paramètre permettent d'appeler les fonctions avec plus de souplesse, sans que l'utilisation des noms soit une obligation.

Un autre exemple :

```
def log(message=None) :
    if message:
        print("{0}".format(message))
```

- On peut appeler cette fonction avec un argument (qui peut être None).

```
log("Fichier fermé")
# Fichier fermé
log(None)
#
```

- Mais on peut l'appeler sans argument auquel cas, la valeur par défaut joue son rôle (ici None) :

```
log()
#
```

XXI-1 Attention aux paramètres mutables

```
def function(item, stuff = []):
    stuff.append(item)
    print(stuff)

function(1)
# Donne 1

function(2)
# Donne 1,2]
# !!!
```

→ La 2e valeur affichée semble étrange.

☞ Remarques sur la valeur de la liste `stuff` et son comportement étrange :

- L'argument à valeur par défaut `stuff` est évaluée une seule fois et ce lors de la définition de la fonction. Ensuite, toute modification de `stuff` se répercute sur l'emplacement initialement allouée à `[]` !!
- Python **ne vérifie pas** si cette valeur par défaut (en fait la case mémoire associée à initialisée à `[]`) a changé et continue de l'affecter à `stuff`. C'est pourquoi au 2e appel, `stuff = [1]`.
- Puisque nous affectons des valeurs différentes à `stuff`, nous changeons à chaque fois sa valeur par défaut. Et lorsque nous rappelons cette fonction, nous récupérons la valeur par défaut modifiée.
- Pour éviter cette situation étrange, il est conseillé de NE PAS UTILISER un objet mutable comme argument à valeur par défaut.

- Il est néanmoins possible de corriger ce comportement :

```
def function(item, stuff = None):
    if stuff is None:
        stuff = []
    stuff.append(item)
    print(stuff)

function(1)
# prints '[1]'

function(2)
# prints '[2]', as expected
```

☞ Ce comportement semble bizarre car normalement, `stuff` contient `[1]` après le 1er appel. Pourtant, le test sur `stuff` fait au début est pour savoir si on a appelé la fonction sans donner ce paramètre (et `None` n'est pas mutable). Donc, le test (`if stuff is None`) réussit et nous permet de savoir si le 2e paramètre a été fourni ou non.

XXII Nombre variable de paramètres d'une fonction

- Un exemple :

```
def do_something(a, b, c, *args):
    print(a, b, c, args)

do_something(1,2,3,4,5,6,7,8,9)
# Donne '1, 2, 3, (4, 5, 6, 7, 8, 9)'
```

- On peut également avoir une nombre quelconque de clé-valeur (voir les *Dicts*) comme paramètre. Une fois qu'on aura défini tous les autres paramètres, on utilise une variable précédée de `**` : soit `**V`. Python utilisera tout le reste des paramètres clé-valeurs et les place dans une dict et l'affecte à `V`.

```
def do_something_else(a, b, c, *args, **kwargs):
    print(a, b, c, args, kwargs)

do_something_else(1,2,3,4,5,6,7,8,9, timeout=1.5)
# Donne '1, 2, 3, (4, 5, 6, 7, 8, 9), {'timeout': 1.5}'
```

- Un problème se pose dans les paramètres par défaut et paramètres nommés. On définit une fonction et des paramètres nommés.

```
def do_something(a, b, c, afficher = True, *args):
    if afficher:
        print(a, b, c, args)

do_something(1, 2, 3, 4, 5, afficher = True)
# ERREUR : TypeError: do_something() got multiple values for paramètre 'afficher'

do_something(1, 2, 3, afficher = True, 4, 5, 6)
# ERREUR SyntaxError: non-keyword arg after keyword arg
```

- Le problème : il n'y a pas de moyen pour spécifier `afficher` comme un paramètre clé-valeur tout en fournissant des paramètres autres.

→ La seule solution pour passer `afficher` ici est d'utiliser un paramètre non clé-valeur.

```
do_something(1, 2, 3, True, 4, 5, 6)
# Résultat : 1, 2, 3, (4, 5, 6)
```

XXII-1 Un autre exemple

```
def Test1(*args, **kwargs):
    print('Test1')
    for arg in args:
        print(arg)
    for k, v in kwargs.items():
        print(k, '=', v)
    print()

"""
Test1(1,2,3)
Test1
1
2
3

Test1(1,b=3)
Test1
1
b = 3

Test1(a=1,b=3)
Test1
b = 3
a = 1

Test1(a=4, 1,b=3)
File "<ipython-input-137-f14bd18d0943>", line 1
Test1(a=4, 1,b=3)

SyntaxError: non-keyword arg after keyword arg

Test1(1,a=4, b=3)
Test1
1
b = 3
a = 4
"""
```

XXII-2 Passer des paramètres par défaut d'une fonction à une autre

- Un exemple :

```
def func(foo, bar=None, **kwargs):
    pass # On définira cette fonction plus tard !

....

#Appel :
desVars=1,2,3
func(42, bar=314, extra=desVars) # extra est un mot réservé
```

- Un autre exemple : on réceptionne les paramètres en utilisant * et ** dans les paramètres de la fonction. On peut passer ces paramètres comme des paramètres d'appel d'une autre fonction.

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

XXIII Complément : Listes en Compréhension

• On lit parfois (par erreur) *la compréhension de liste*. Notons que lorsqu'on construit une liste élément par élément (via `append()` ou une liste donnée explicitement comme `[a,b,c,...]`), on appellera cela une *liste par extension*. *Intension* et *Compréhension* ont le même sens.

• La *liste en compréhension* souligne l'aspect fonctionnel de Python, en particulier avec *map*, *filter* et *reduce* et les *expressions-lambda*.

• Ce mécanisme procure un moyen rapide et compact pour créer des listes. La *liste en compréhension* est davantage préconisée en Python 3.4 à cause des modifications sur *map* et *filter* (qui sont devenus des *générateurs*).

• La liste en compréhension permet une traduction quasi directe des *intensions*. Soit :

$$S = \{x^2 : x \in \{0 \dots 9\}\}$$

$$V = (1, 2, 4, 8, \dots, 2^{12})$$

$$M = \{x | x \in S \wedge \text{even}(x)\} \quad \text{avec } \text{even}(x) : x = 2k, k \in \mathbb{N}$$

Qu'on écrive en Python :

```
S = [x**2 for x in range(10)]
V = [2**i for i in range(13)]
M = [x for x in S if x % 2 == 0]

print(S); print(V); print(M)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
# [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
# [0, 4, 16, 36, 64]
```

☞ On a demandé les résultats sous forme de listes. En utilisant des `()` à la place des `[]`, on obtiendrait ici un *générateur* (mais ce ne sera plus une *liste* en compréhension !). Notons également que `{}` sont utilisées pour les *Sets* et les *Dicts* et les `'()'` également pour les *tuples*.

• On peut mieux comprendre les listes en compréhension à travers leurs *fonctions* équivalentes :

```
S = []
for i in range(10):
    S.append(i*2)

# Et
S = [i*2 for i in range(10)]

S
# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

• Ou encore

```
M = []
for i in range(10):
    if i % 2 == 0:
        M.append(i)

# Et
M = [i for i in range(10) if i % 2 == 0]

M
# [0, 2, 4, 6, 8]
```

XXIII-1 Exemple : Nombres premiers

• Pour construire la liste des nombres premiers dans l'intervalle 1..50, on construit une liste de nombres non premiers puis on utilise son complément pour obtenir des nombres premiers :

```
noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
primes = [x for x in range(2, 50) if x not in noprimes]
print(primes)
# [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

→ La valeur 8 correspond à $\text{int}(\sqrt{50}) + 1$.

XXIII-2 Exemple : Manipulation de caractères

- Manipulation de listes de caractères :

```
[c for c in "foobar"]
# Donne ['f', 'o', 'o', 'b', 'a', 'r']
```

→ L'expression `[c for c in "foobar"]` est, du fait de la présence des `[]`, équivalente à `list("foobar")`.

- Une autre liste en compréhension pour construire une liste des caractères :

```
texte = "La meilleure façon de marcher..."
[texte[i] for i in range(len(texte))]

# Donne : ['L', 'a', ' ', 'm', 'e', 'u', 'r', 'e', ' ', 'f', 'a', 'ç', 'o', 'n', ' ', 'd', 'e', ' ', 'm', 'a', 'r', 'c', 'h', 'e', 'r', '...']
```

D'autres exemples :

```
liste_origine = [0, 1, 2, 3, 4, 5]
[nb * nb for nb in liste_origine]
#==> [0, 1, 4, 9, 16, 25]

# Lire 10 fois le clavier et faire une eliste :
L=[input("une chose") for i in range(10)]

# Copie d'une liste
ll=[1,2,3]
inv=[ll[i] for i in range(3)]
```

- L'expression de la liste en compréhension peut contenir des filtres (tests, appels de fonctions, etc) :

```
liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[nb for nb in liste_origine if nb % 2 == 0]
# ==> [2, 4, 6, 8, 10]
#
def pas_multiple_de_6(x): return x % 2 != 0 and x % 3 != 0
[i for i in range(2, 25) if pas_multiple_de_6(i)]
# Donne : [5, 7, 11, 13, 17, 19, 23]
#
def cube(x): return x*x*x
[cube(i) for i in range(1, 11)]
#Donne : [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

- En Python 3, la liste en compréhension vient remplacer *map* et *filter* (voir plus loin).

XXIII-3 Exercice Simple

- Dans cet exercice, on étudie les deux formes de construction des listes.
 - Utilisez une construction classique puis une liste en compréhension pour
 - ajouter 3 à chaque élément d'une liste d'entiers de 0 à 5.
 - ajouter 3 à chaque élément d'une liste d'entiers de 0 à 5, mais seulement si l'élément ≥ 2 .
 - obtenir la liste `['ad', 'ae', 'bd', 'be', 'cd', 'ce']` à partir des chaînes "abc" et "de".
 - calculer la somme d'une liste d'entiers de 0 à 9.
- Indication : utilisez deux boucles `for` imbriquées.

XXIII-4 Exercice : Mots en Majuscule

- Soit la chaîne `mots = 'Le petit chien jaune et noir mange la soupe chaude'`

Extraire les mots de cette phrase (utiliser la fonction `str.split()`) et transformez et affichez ces mots en majuscule et donnez leur taille. Pour obtenir un mot *M* en majuscule / minuscule, vous pouvez utiliser `M.upper()` / `M.lower()`.

XXIII-5 Exercice : anagramme

- Écrire le code qui vérifie si deux strings sont des anagrammes.

XXIII-6 Bonus : tousDiff

- Créer une liste d'entiers avec un tirage aléatoire (de valeurs entre 0 et $3N$, $N = 10$ par exemple) puis tester si les éléments de cette liste sont deux à deux différents.

XXIV Manipulation de fichiers

XXIV-1 Exemple

- Pour simplement ouvrir un fichier avec contrôle :

```
import os
os.chdir("/home/alex/Telechargement/Apprendre-Python-3")
try:
    f = open('fic-test.txt')
except OSError as err:
    print("OS error: {0}".format(err))
    raise # Pour ne pas continuer ci-dessous.
        # Utiliser cela au lieu de 'exit()' qui arrête l'interprète Python.

# Si le fichier n'existe pas, on a l'erreur (exception) :
# OS error: [Errno 2] No such file or directory: 'fic-test1.txt'

# Et 'raise' cela donnera
# FileNotFoundError: [Errno 2] No such file or directory: 'fic-test1.txt'
```

- Un autre exemple : avec lecture d'entiers

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())

except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Conversion en entier impossible.") # Pour l'erreur de conversion
except:
    print("Erreur ? : ", sys.exc_info()[0]) # Autre erreur
    raise # On relève l'exception pour le niveau englobant.
```

XXIV-2 Rappel sur la fonction split()

- Cette fonction sépare une chaîne de caractères en liste de mots. Elle utilise comme séparateur par défaut les caractères blancs (espace ou tabulation). Elle supprime tous les séparateurs contigus. Le nombre de séparation maximum est indiqué par un deuxième paramètre optionnel.

Exemples :

```
# split()
a="Cette phrase, devient,, une\liste"
b=a.split()
print(b)
# ['Cette', 'phrase,', 'devient,,', 'une', 'liste']
# '\t' symbolise une tabulation.

# split(sep)
a="Cette phrase, devient,, une\liste"
b=a.split(",")
print(b)
# ['Cette phrase', ' devient', 'une\liste']

# split(chaîne-sep) avec une chaîne comme séparateur
# split() avec une chaîne comme séparateur
a="Cette phrase, devient,, une\liste"
b=a.split(",,")
print(b)
# ['Cette phrase, devient', ' une\liste']

# split() avec maximum de séparations
a="Cette phrase, devient,, une\liste"
b=a.split(None,3)
print(b)
# ['Cette', 'phrase,', 'devient,,', 'une\liste']
```

- `splitlines()` sépare un texte en liste de lignes. Un exemple :

```
a="Ligne1- a b c d e f\nLigne2- a b c\n\nLigne4- a b c d"
b=a.splitlines()
print(b)

# [ 'Ligne1- a b c d e f', 'Ligne2- a b c', '', 'Ligne4- a b c d' ]
```

☞ Un truc :

Si vous voulez lire une série d'entiers sur une ligne :

```
serie = map(int, input().strip().split(" "))
```

XXIV-3 Un exemple avec with

- Exemple (vu plus haut dans la partie système ?, chercher "with")

```
f = open("fic-test.txt")
f
#<_io.TextIOWrapper name='fic-test.txt' mode='r' encoding='UTF-8'>

f.__enter__()      # La méthode de la classe
#<_io.TextIOWrapper name='fic-test.txt' mode='r' encoding='UTF-8'>

f.read(1)          # Lire un caractère
# 'E'

f.__exit__(None, None, None)
f.read(1)
# ValueError: I/O operation on closed file
```

Avec **with**, c'est plus simple : pour ouvrir le fichier, traiter son contenu (ça, c'est nous qui le faisons !) et de le fermer à la fin.

```
with open("fic-test.txt") as f:
    data = f.read()
    # traiter les données, par exemple :
    print(data)
```

- Créer avec **with** : `hello.txt` va exister (ou écrasé s'il existait)

```
with open("hello.txt", "w") as f:
    f.write("Hello World")
```

- with** et **split()** :

```
with open('data.txt', 'r') as f:
    data = f.readlines()

for line in data:
    words = line.split()
    print(words)
    # Et si vous voulez chaque caractère de 'words', faites comme pour les listes.
```

XXIV-4 Un exemple utilisant les générateurs et les listes en compréhension

- Dans cet exemple, le fichier '`fic-test.txt`' contient :

```
Exercice 25 : cet exercice concerne les fichiers.
Lire un fichier contenant des nombres et des lignes vides et extraire :
- Les lignes
- Les mots de chaque ligne

- Les lettres de chaque mot
- Extraire les nombres
- Compter le nombre de phrases (terminées avec '.')
- etc.

Voire aussi l'exercice 561.
```

- On fera attention aux générateurs qui sont des ressources épuisables.

```
f = open('fic-test.txt')

# Un ggénérateur
lignes_non_vides = (line for line in f if line.strip())
mot = "Extraire"

# Un générateur
lignes_qui_contiennent_mot = (l for l in lignes_non_vides if mot in l)
print(lignes_qui_contiennent_mot)
# <generator object <genexpr> at 0x7f291d512ee8>

# Un générateur
lignes_qui_ne_finissent_pas_par_un_point = (l for l in lignes_qui_contiennent_mot if not l.endswith('.') )

# On récupère la liste associée à un générateur
lignes_numerotees = list(enumerate(lignes_qui_ne_finissent_pas_par_un_point))
print(lignes_numerotees)

#[(0, '- Extraire les nombres \n')]

f.close()
```

XXIV-5 Création d'un fichier

- Un exemple simple :

```
file = open("newfile.txt", "w")
file.write("hello world in the new file\n")
file.write("and another line\n")
file.close()
```

☞ Remarques :

- Noter bien que `readline` renvoie la chaîne vide `"` après **EOF**, pas d'exception.
- On peut essayer en testant `vs "` ou utiliser **`strip()`** pour déceler les lignes vides : `ligne.strip()` réussit si *ligne* n'est pas vide.
- Noter que `ligne=readline()` conserve `'\n'`.
- Pour trouver un `'.'` à la fin de ligne (qui contient `'\n'`), on peut tester `if ligne.endswith('.')`.

XXIV-6 Exercice

- Lire un fichier et extraire :
 - La fréquence des mots
 - La fréquence des lettres
 - Le nombre de phrases (terminant avec un `'.'`)
 - Les nombres
- Le contenu du fichier (*test_fic.txt*) a été donné ci-dessus.

```
% Exercice 25 : cet exercice concerne les fichiers.
% Lire un fichier contenant des nombres et des lignes vides et extraire :
% - Les lignes
% - Les mots de chaque ligne
%
% - Les lettres de chaque mot
% - Extraire les nombres
% - Compter le nombre de phrases (terminées avec '.')
% - etc.
%
% Voir aussi l'exercice 561.
%
```

XXV Compléments sur la mise en page

XXV-1 format pour les conversions

- Conversion de string en binaire (voir la fonction `format()` en section [XIII-1](#) page 22. L'exemple suivant a été vu plus haut.

```
st = "hello Debase"
''.join(format(ord(x), 'b') for x in st)
# '1101000 1100101 1101100 1101100 1101111 100000 1000100 1100101 1100010 1100001 1110011 1100101'
```

→ Le passage par la fonction `ord(x)` est ici obligatoire. `ord(x)` renvoie un entier pour un caractère contenant un chiffre.

Si on doit nommer le range du champ à formater (ici `0:b`) :

```
st = "hello Debase"
print(''.join(["{:b}".format(ord(x)) for x in st]))
# 1000101 1100011 1101100
```

- Conversion de string en hexadécimal :

```
s = 'Hello Debase of Ecl'
''.join(format(ord(x), 'x') for x in s)
# '48 65 6c 6c 6f 20 44 65 62 61 73 65 20 6f 66 20 45 63 6c'
```

XXV-2 Justification avec `rjust`, `ljust` et `center`, bourrage avec `zfill`

- Justification des strings à droite ou à gauche, centrage ou remplissages par des zéros :

```
s = "Python"
s.center(10)                # centrer sur 10 positions
# Donne --> ' Python '

s.center(10, "*")           # centrer sur 10 positions en complétant par des '*'
# Donne --> '**Python**'

s = "Training"
s.ljust(12)                 # justifier à gauche sur 12 positions
# Donne --> 'Training '

s.ljust(12, ":")
# Donne --> 'Training :::'

s = "Programming"
s.rjust(15)
# Donne --> ' Programming '
s.rjust(15, "~")
# Donne --> '~~~~Programming'

account_number = "43447879"
account_number.zfill(12)
# Donne --> '000043447879'

# Même chose avec rjust :
account_number.rjust(12, "0")
# Donne --> '000043447879'
```

☞ En fait, `str.ljust(n)` construit une nouvelle chaîne de taille `n` à partir de `str` en plaçant `str` à gauche de cette nouvelle chaîne. De même pour `str.rjust(n)` et `str.center(n)`.

XXV-3 str() et repr()

- `str()` permet d'obtenir un string lisible (par l'humain) alors que `repr()` permet d'obtenir une représentation lisible par l'humain et par l'analyseur de Python.
- La plupart du temps, ces deux fonctions donnent le même résultat. Cependant, `repr()` s'applique dans les situations (objets, construction complexes) où `str()` pourrait ne pas suffire (voir ci-dessous).

- Exemples :

```
s = 'Hello, world.'
str(s)
# Donne --> 'Hello, world.'

repr(s)
# Donne --> "'Hello, world.'"
str(1/7)
# Donne --> '0.14285714285714285'

x = 10 * 3.25
y = 200 * 200
s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
print(s)
# Donne --> The value of x is 32.5, and y is 40000...

# Le repr() d'un string ajoute des quotes et backslashes:
hello = 'hello, world\n'
str(hello)
# Donne --> 'hello world\n'
repr(hello)
# Donne --> 'hello, world\\n' noter le double slash

# L'argument d'appel de repr() peut être n'importe quel objet Python :
repr((x, y, ('spam', 'eggs'))))
# Donne --> "(32.5, 40000, ('spam', 'eggs'))"
```

- Un exemple de mise en colonne de valeurs : afficher en colonnes $x, x^2, x^3, x \in 1..10$

```
for x in range(1, 11):
    print(repr(x).rjust(2), repr(x*x).rjust(3), repr(x*x*x).rjust(4))
# Donne -->
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

☞ Remarques : Si vous souhaitez terminer chaque écriture par autre chose qu'un retour charriot, utilisez l'argument `end=...` dans l'appel de `print`. Par défaut, `end='\n'`.

Par exemple :

```
for x in range(1, 6):
    print(repr(x).rjust(2), end='#'*x+'\n')    on ajoute x fois # + \n à la fin de chaque ligne

# Donne -->
1#
2##
3###
4####
5#####
```

- L'intérêt de `end=...` se trouve souvent dans les cas où on a deux ordres `print` et on voudrait que le deuxième soit à la suite du premier.

```
print(1,2, end=' ')    # Ne pas passer à la ligne pour que ...
x=fonction_qui_calcule(3)    # ... la
print(x)    # ... valeur de x s'affiche à la suite sur la même ligne
```

XXV-4 Un truc sur print

- Si on doit répéter un format d'écriture (par exemple %2d) pour k valeurs, on peut répéter ce format par "%2d"*8. Par exemple :

```
# Création de 8 valeurs
a, b, c, d, e, f, g, h = range(8) # assigner 0..7 à a, b, c, d, e, f, g, h
X = (a, b, c, d, e, f, g, h)

print("%2d"*8 % X)
# 0 1 2 3 4 5 6 7
```

- Remarquer qu'on aurait pu, dans ce cas précis, écrire :

```
a, b, c, d, e, f, g, h = range(8) # assigner 0..7 à a, b, c, d, e, f, g, h
X = (a, b, c, d, e, f, g, h)

[print("%2d" % i) for i in X]
```

XXVI Gestion du temps

```
import timeit
timeit.timeit("x = 2 + 2")
# 0.188.....
timeit.timeit("x = sum(range(10))")
# 5.589.....
```

- Pour une exécution en ligne de commande d'une fonction `my_fonction()` que l'on aurait définie dans le module `my_module` et mesurer le temps, on peut écrire :

```
$ python3.4 -m timeit -s "import my_module as my_mo_sa" "my_mo_sa.my_fonction()"
```

- On peut également utiliser la fonction `time.time()` pour mesurer le temps d'exécution d'une fonction :

```
from time import time

def count():
    deb = time()
    [x for x in range(10000000) if x%4 == 1]
    fin = time()
    print(fin-deb)

count()
# 1.213
```

- Il est conseillé d'utiliser `timeit` (exemple ci-dessus) à la place de `time` car `timeit` est plus juste et tient compte du ramassage des miettes (gc : garbage collector).

Table des Matières

I. Modalités des travaux à rendre	1
II. Objectifs de ce BE	2
II-1. Principe de ce BE	2
II-2. Documentation.	2
II-3. A savoir	2
III. Introduction.	3
III-1. Commentaires en Python	3
IV. Exemples et exercices simples	4
IV-1. Exemple 1 : moyenne.	4
IV-2. Exemple 2 : racines carrées	4
IV-3. Exercice : pair/impair	5
V. Introduction aux fonctions	5
V-1. Exemple : moyenne de deux entiers	5
V-2. Exercice : Racines d'une quadratique avec une fonction	5
V-3. Exercice : Moyenne et écart type	5
V-4. Variante : Exercice (suite écart type)	6
VI. Créer un script Python indépendant	6
VI-1. Une façon simple	6
VI-2. Une solution plus élaborée	7
VII. Types principaux en Python	8
VII-1. Exemples	8
VIII. Conversions de types.	8
VIII-1. Exemples de conversion de types.	9
VIII-2. Exercice	9
IX. Modules Python	10
X. La fonction range	10
X-1. Deux ou trois trucs pratiques sur range.	12
X-2. Bonus : sommes	12
XI. Les listes	13
XI-1. Un exemple sur les listes	14
XI-2. Autres exemples de fonctions sur les listes.	15
XI-3. Itération à l'envers	16
XI-4. Itération sur une liste avec enumerate	17
XI-5. Exemple : Pile	17
XI-6. Bonus : TDA Queue	18
XI-7. Exercice : enlever les doublons d'une liste	18
XI-8. Bonus : Scinder une liste.	18
XII. Tuples.	19
XII-1. Accès aux éléments d'un tuple	19
XIII. Introduction à la mise en page.	20
XIII-1. La fonction format	22
XIII-2. Exercice : table de x , x^2x , \sqrt{x} .	23
XIV. Itérations : While, for	23
XIV-1. While : Celcius-Fahrenheit	23
XIV-2. For : Celcius-Fahrenheit	23
XIV-3. Exercice : valeur de e	25
XIV-4. Exercice : nombre premier	25
XIV-5. Bonus : Nombre Premier par diviseurs propres.	25
XIV-6. Bonus : Diviseurs des nombres Parfaits et Premiers	25
XIV-7. Bonus (suite) : Importer vos fonctions.	25
XIV-8. Exemple : compter les lettres.	26
XIV-9. Exercice : compter les voyelles et consonnes	26
XIV-10. Exercices : lancer des dés	27
XIV-11. Bonus : Tri sélection.	27
XV. Les exceptions	27
XV-1. Exemple 1	27
XV-2. Exemple : Min/Max d'une séquence à la volée	28
XV-3. raise : deux exemples.	29
XV-4. Exercice : moyenne et fonction.	30
XV-5. Expression Pass	30
XVI. Génération de nombres aléatoires.	30
XVI-1. Exemple : indice du minimum d'une liste aléatoire d'entiers	31
XVI-2. Exemple : amplitude et la moyenne d'une liste aléatoire de réels	31
XVI-3. Échantillonnage : Normale et Uniforme.	32
XVI-4. Complément : Tirage avec remise.	32
XVI-5. Exemple : Tirage de cartes	33
XVI-6. Exercice : Tirage de cartes	34
XVI-7. Bonus : Tirage de cartes (bis)	34
XVI-8. Bonus : Fréquences de dés	34
XVII. Quelques fonctions sur les chaînes	34
XVIII. Imbrication de fonctions	35
XVIII-1. Quelques règles utiles	35
XVIII-2. Bonus : Tri par insertion	37
XVIII-3. Bonus : Tri split-merge	38

XIX. Complément sur la Visibilité des variables	39
XIX-1. Variables locales	39
XIX-2. global	40
XIX-3. nonlocal	40
XIX-4. Un exemple récapitulatif	41
XX. Introduction à la récursivité	42
XX-1. Exemple : Tri bulle récursive	42
XX-2. Un exemple de récursivité indirecte	42
XX-3. Exercice : recherche Dichotomique dans une liste triée	43
XX-4. Bonus : Tri par insertion	43
XX-5. Bonus : Tri par insertion utilisant la recherche dichotomique	43
XX-6. Bonus : lancers de dés	43
XXI. Paramètre formel à valeur par défaut des fonctions	44
XXI-1. Attention aux paramètres mutables	44
XXII. Nombre variable de paramètres d'une fonction.	45
XXII-1. Un autre exemple	46
XXII-2. Passer des paramètres par défaut d'une fonction à une autre.	46
XXIII. Complément : Listes en Compréhension	47
XXIII-1. Exemple : Nombres premiers	47
XXIII-2. Exemple : Manipulation de caractères	48
XXIII-3. Exercice Simple	48
XXIII-4. Exercice : Mots en Majuscule	48
XXIII-5. Exercice : anagramme	48
XXIII-6. Bonus : tousDiff	49
XXIV. Manipulation de fichiers.	49
XXIV-1. Exemple	49
XXIV-2. Rappel sur la fonction split()	49
XXIV-3. Un exemple avec with	50
XXIV-4. Un exemple utilisant les générateurs et les listes en compréhension	50
XXIV-5. Création d'un fichier	51
XXIV-6. Exercice	51
XXV. Compléments sur la mise en page	52
XXV-1. format pour les conversions	52
XXV-2. Justification avec rjust, ljust et center, bourrage avec zfill	52
XXV-3. str() et repr()	53
XXV-4. Un truc sur print	54
XXVI. Gestion du temps	54