

# 01-cours\_python\_variables\_print\_input

January 11, 2020

## 1 Prise en main de l'environnement, variables, interactions avec l'utilisateur

### 1.1 Objectifs

Le but de cette séance est d'exécuter quelques commandes simples et de se familiariser avec quelques notions de base d'algorithmique.

### 1.2 Les différentes versions de Python

- Python 2.7 (ne sera plus supportée fin 2019)
- Python 3.x (actuellement Python 3.7)

### 1.3 Pour bien s'organiser...

#### 1.3.1 Quelques notions

En programmation informatique pour les noms de fichiers (mais aussi pour les noms de variables, de fonctions...) : - Pas d'accent - Pas d'espaces

#### 1.3.2 Mettre en place son organisation avec des répertoires

- Choisir un répertoire de son ordinateur
- Dans ce dernier, créer un répertoire cours
- Dans ce dernier, créer un répertoire python
- Dans ce dernier, créer un répertoire "seance1"

#### 1.3.3 Les fichiers Python

- Chaque programme écrit sera enregistré dans un fichier qui portera un nom avec l'extension ".py".
- Pour chaque exercice proposé, vous l'enregistrerez dans le répertoire "seance1" avec le nom qui vous convient, par exemple *exercice1.py* (ou *exercice1\_affectation\_variable.py* si vous préférez un nom plus parlant).

### 1.3.4 Vue d'ensemble

**Note : ce ne sont pas des conseils, ce sont des obligations !**

## 1.4 Utilisation d'un environnement de développement (IDE) : Spyder

Pour écrire un programme en Python, on peut (mais ce n'est pas obligatoire !) utiliser un environnement de développement (IDE en anglais) qui permet notamment d'afficher les mots-clefs en couleur, de consulter la valeur d'une variable, de bénéficier de l'auto-complétion... Celui que nous utiliserons répond au nom de *Spyder*.

Dans une fenêtre de l'éditeur Spyder, taper le magnifique programme suivant :

```
# Cool, this is my first python script!  
print("Hello world!")
```

- L'enregistrer dans le répertoire : “cours - python - seance1” et le nommer *hello\_world.py*
- L'exécuter...
- Noter également qu'en plus de la fenêtre principale, vous avez également une fenêtre nommée “console IPython”. Elle peut être très pratique pour effectuer des tests rapides de petits programmes (Python).

## 1.5 Les variables

### 1.5.1 Présentation

- Une variable est une référence vers une zone de contenu (adresse mémoire de l'ordinateur). C'est donc une façon de nommer un contenu pour pouvoir le retrouver ultérieurement.
- Lorsqu'on exécute des calculs, qu'on appelle des fonctions ou qu'on effectue divers traitements, il est souvent intéressant de pouvoir en conserver le résultat. C'est tout le principe des variables : garder la trace d'une valeur.
- Les noms de variables sont des identificateurs arbitraires, de préférence assez courts tout en étant aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée référencer (la sémantique de la donnée référencée par la variable).

### 1.5.2 Les noms de variables (détail)

- Une variable ne peut être composée que de lettres minuscules ou majuscules, de chiffres et du symbole souligné “\_”.
- Une variable ne peut commencer par un chiffre.
- De plus le langage Python est sensible à la casse (les majuscules et les minuscules). Ainsi `var1`, `Var1` et `VAR1` sont des variables différentes.
- Ne jamais utiliser de mot clé réservé comme nom de variable (même si Python l’autorise). Exemple: *list*, *str*,... Si dans votre éditeur, le nom de votre variable apparaît en couleur, posez-vous des questions...
- Ne jamais mettre d’accents dans les noms de variables.
- On utilisera toujours des noms de variables explicites (*toto*, *var* etc. sont à proscrire).
- Conventions de nommage:
  - `ma_variable`
  - `MA_CONSTANTE`
  - `ma_fonction`
  - `MaClasse`

Ces conventions de nommages ne sont pas obligatoires mais recommandées. Elles sont régies par la [PEP8](#) (Python Extension Proposals).

### 1.5.3 Affectation

L’**instruction** d’affectation est notée `=` en Python : `nom = valeur`. Le nom de la variable à modifier est placé dans le membre de gauche du signe `=`, la valeur qu’on veut lui attribuer dans le membre de droite.

Ainsi, les deux membres du signe `=` ne jouent pas le même rôle. Le membre de droite de l’affectation est d’abord **évalué** sans être modifié. Puis la valeur obtenue est affectée à la variable dont le nom est donné dans le membre de gauche de l’affectation. Cette opération ne modifie que le membre de gauche de l’affectation.

Le membre de droite d’une affectation peut être une **constante** ou une **expression** évaluable.

```
[3]: # Exemple d'affectation d'une variable
    x = 1
```

```

y = 1 + 2/3
z = y - x
z = z + 4  # On peut aussi écrire en Python z += 4
print(z)  # L'instruction print permet d'effectuer un affichage à l'écran

```

4.6666666666666666

Notons que, pour utiliser une variable dans une expression, elle doit avoir été préalablement définie:

```

[8]: # Programme correct
x = 1
y = x

```

```

[9]: # Programme incorrect
z = t + 3

```

```

      □
↳ -----

NameError                                Traceback (most recent call↳
↳last)

<ipython-input-9-2539b134b630> in <module>
      1 # Programme incorrect
----> 2 z = t + 3

NameError: name 't' is not defined

```

#### 1.5.4 Exercices

##### Instruction versus expression

Parmi les lignes ci-dessous indiquer quelles sont des instructions et quelles sont des pures expressions :

- $1 + 3$
- $x = 2$
- $\text{print}(x)$
- $\cos(x)$
- $x$
- $y = \sin(x)$
- $z = x + 2$
- $4 = p$

Correction :

- $1 + 3$  : expression

- $x = 2$  : instruction
- $\text{print}(x)$  : instruction
- $\cos(x)$  : expression
- $x$  : expression
- $y = \sin(x)$  : instruction
- $z = x + 2$  : instruction
- $4 = p$  : incorrect !

### Exécuter un algorithme dans sa tête

Sans utiliser Python, indiquer le contenu de la variable  $c$  à la fin du programme ci-dessous :

```
a = 1
b = 3
c = a - b
c = c + 3
```

### Calcul de la moyenne

Soient quatre nombre : 8, 20, 30 et 2. En utilisant six variables, calculer la moyenne et l'afficher.

```
[4]: n1 = 8
      n2 = 20
      n3 = 30
      n4 = 2
      somme = n1 + n2 + n3 + n4
      moyenne = somme / 4
      print(moyenne)
```

15.0

### Un grand classique de la programmation : l'échange de variables

Soient deux variables  $x = 1$  et  $y = 2$ . On souhaite avoir à la fin du programme  $x = 2$  et  $y = 1$ . Le professeur de Python a commencé à écrire un programme mais il ne fonctionne pas, il faut donc l'aider :

```
[5]: x = 1
      y = 2
      y = x
      x = y
      print(x)
      print(y)
```

1  
1

```
[6]: # Première version : utiliser une variable temporaire
      x = 1
      y = 2
      tmp = y
      y = x
```

```
x = tmp
print(x)
print(y)
```

2  
1

[7]: *# Deuxième version : utiliser les "facilités" du langage Python*

```
x = 1
y = 2
x, y = y, x
print(x)
print(y)
```

2  
1

### A la librairie

Un libraire propose une réduction de 3.5% sur le prix hors taxes (HT) d'un livre à 12.35 € HT. Sachant que la taxe sur la valeur ajoutée (TVA) sur les livres est de 5.5%, écrire un programme qui permettra d'afficher le prix au client.

[23]: *# Initialisation des variables*

```
prix_ht = 12.35
tva = 5.5
taux_reduction = 3.5

# Calcul de la taxe et de la réduction
reduction = prix_ht * taux_reduction / 100
prix_ht_reduit = prix_ht - reduction
taxe = prix_ht_reduit * tva / 100

# Calcul du résultat final
prix_vente = prix_ht_reduit + taxe
print(prix_vente)
```

12.57322625

### 1.5.5 Le typage dynamique

Jusqu'à présent nous n'avons manipulé que des variables de type entier ou décimal. Le fait d'exécuter une instruction d'affectation définit en même temps le type de la variable. Exemples :

```
nombre_entier = 22
nombre_decimal = 12.5
chaine_caracteres = "Bonjour" # Ou 'Bonjour' cela est équivalent
```

C'est ce qu'on appelle le **typage dynamique**. Notez que cela n'est pas valable pour tous les langages informatiques. C'est une particularités de Python.

**Les types de base** Les types de base (*non mutables*) de variables sont :

- ***int*** pour les entiers,
- ***float*** pour les décimaux (nombres flottants)
- ***str*** pour les chaînes de caractères.
- ***bool*** pour les booléens

Pourquoi le type est-il important ? C'est lui qui va déterminer le type d'opération qu'on va pouvoir effectuer ainsi que le résultat :

```
[10]: # Opérations sur les nombres
a = 1
b = a + 2 # Entier + Entier
print(b)
c = 1.4
d = b + c # Entier + Décimal
print(d)
```

3  
4.4

```
[11]: # Opérations sur les chaînes de caractères
prenom = "Wolfgang Amadeus"
nom = "Mozart"
separateur = " " # C'est un espace
nom_complet = prenom + separateur + nom # Concaténation de chaînes
print(nom_complet)
```

Wolfgang Amadeus Mozart

Il n'est pas possible de mixer certaines opérations si les types sont différents :

```
[14]: a = 1
b = 2
c = a + b # OK -> 3
d = "1"
e = "2"
f = d + e # OK -> "12"
print(c)
print(f)
g = a + e # Erreur
```

3  
12

↳ -----

```
TypeError                                Traceback (most recent call_
↳last)
```

```
<ipython-input-14-c3ff953c1c7b> in <module>
      7 print(c)
      8 print(f)
----> 9 g = a + e  # Erreur
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Pour effectuer des opérations entre variables de types différents, il faut effectuer une **conversion explicite** :

```
[17]: # Conversion str -> int
a = 1
b = "2"
c = a + int(b)
print(c)
# Conversion int -> str
chaine = "Nombre = "
valeur = 12
resultat = chaine + str(valeur)
print(resultat)
```

```
3
Nombre = 12
```

On peut utiliser la *fonction* Python `type()` pour connaître le type d'une variable en Python

```
[12]: nom = "Mozart"
print(type(nom))
age = 78
print(type(age))
condition = True
print(type(condition))
```

```
<class 'str'>
<class 'int'>
<class 'bool'>
```

Remarque: dans un programme, même si **ce n'est pas recommandé** pour des questions de lisibilité, grâce au typage dynamique, il est possible qu'un même nom de variable prenne plusieurs types :

```
[16]: variable_fourre_tout = "toto"
print(type(variable_fourre_tout))
variable_fourre_tout = 123
```



```
print(type(variable_fourre_tout))
```

```
<class 'str'>
```

```
<class 'int'>
```

### 1.5.6 Exercices

#### Construction d'une chaîne de caractères

Afficher le message suivant "La voiture roule à 125.5 km/h."

Utiliser :

- Deux variables de type chaîne (str)
- Une variable de type décimal (float).

On utilisera également une variable de type chaîne pour stocker le résultat avant de l'afficher (avec la fonction *print*).

```
[19]: debut_phrase = "La voiture roule à "  
vitesse = 125.5  
fin_phrase = " km/h."  
resultat = debut_phrase + str(vitesse) + fin_phrase  
print(resultat)
```

La voiture roule à 125.5 km/h.

### 1.5.7 Synthèse

L'affectation d'une variable a donc pour effet de réaliser plusieurs opérations en mémoire :

- Créer et mémoriser une valeur particulière
- Créer et mémoriser un nom de variable et établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante
- Attribuer un type déterminé à la variable

## 1.6 Fonctions d'entrées/sorties

### 1.6.1 La fonction *print*

La fonction *print(...)* permet d'afficher à l'écran un message passé en **argument** (ou en paramètres) - c'est-à-dire entre les parenthèses. Voici quelques exemples :

```
[21]: # Avec un seul argument (une seule chaîne de caractères)  
print("Bonjour")  
  
# Avec un seul argument et des variables  
age = 20  
## Cela fonctionne  
print("Vous avez " + str(age) + " ans")  
## Cela ne fonctionne pas  
print("Vous avez " + age + " ans")
```

```
# Cela fonctionne et c'est mieux (plus lisible)
print("Vous avez {0} ans".format(age))
# Cela est encore mieux
print(f"Vous avez {age} ans") # A partir de Python 3.6

# Avec plusieurs arguments (3)...
print("Vous avez", age, "ans")
```

Bonjour

Vous avez 20 ans

Vous avez 20 ans

Vous avez 20 ans

Vous avez 20 ans

### 1.6.2 La fonction *input*

La fonction *input(...)* provoque une interruption de l'exécution d'un programme pour que l'utilisateur puisse entrer une valeur. Le programme reprend son cours quand l'utilisateur appuie sur la touche *Entrée*.

- L'**argument** passé à la fonction *input* est le message qui sera affiché sur l'écran de l'utilisateur.
- La fonction *input* **renvoie** une valeur (qui est celle saisie par l'utilisateur). La plupart du temps on la stockera dans une variable. (Notez au passage que la fonction *print* ne renvoie aucune valeur).

Exemple :

```
[ ]: nom = input("Quel est votre nom ?")
```

**Attention:** la fonction *input(...)* renvoie toujours une chaîne de caractères. Si l'on veut effectuer des opérations mathématiques sur une variable entrée par l'utilisateur (par exemple), il faut préalablement la convertir en effectuant un transtypage (cast, en anglais). Pour les trois types principaux, les transtypes s'effectuent en utilisant les fonctions suivantes : *\* int(...)* *\* float(...)* *\* str(...)*

```
[3]: annee_naissance = input("Entrez votre année de naissance : ")
annee_actuelle = input("Entrez l'année actuelle : ")

# Calcul de l'âge avec conversion
age = int(annee_actuelle) - int(annee_naissance)

print("Vous avez " + str(age) + " ans")
# Instruction équivalente avec conversion implicite
print(f"Vous avez {age} ans")
```

Entrez votre année de naissance : 1999

Entrez l'année actuelle : 2019

Vous avez 20 ans

Vous avez 20 ans

### 1.6.3 Exercice : une calculatrice vraiment basique

Créer un programme qui demande deux entiers a et b à l'utilisateur. Le programme doit calculer puis afficher les résultats suivants :

- la somme de a et de b
- la soustraction de b par a
- le produit de a et de b
- la quotient de b par a
- le reste quand b est divisé par a
- le résultat de a puissance b

```
[1]: valeur_a = input("Valeur de a :") # valeur_a est une chaîne
valeur_b = input("Valeur de b :") # valeur_b aussi
a = int(valeur_a) # a est désormais un entier
b = int(valeur_b) # b aussi
somme = a + b
print(f"Somme de {valeur_a} et {valeur_b} : {somme}")
soustraction = a - b
print(f"Soustraction de {valeur_a} et {valeur_b} : {soustraction}")
multiplication = a * b
print(f"Multiplication de {valeur_a} et {valeur_b} : {multiplication}")
quotient = b / a
print(f"Quotient de {valeur_b} par {valeur_a} : {quotient}")
reste = b % a
print(f"Reste de {valeur_b} par {valeur_a} : {reste}")
puissance = a**b
print(f"{valeur_a} puissance {valeur_b} : {puissance}")
```

```
Valeur de a :12
Valeur de b :3
Somme de 12 et 3 : 15
Soustraction de 12 et 3 : 9
Multiplication de 12 et 3 : 36
Quotient de 3 par 12 : 0.25
Reste de 3 par 12 : 3
12 puissance 3 : 1728
```

### 1.6.4 Exercice : calculer la surface d'une pièce

Ecrire un programme qui demande à l'utilisateur d'entrer successivement la largeur puis la longueur d'une pièce d'une maison. Une fois que les valeurs ont été lues, le programme doit afficher la surface correspondante.

```
[2]: # On utilise la fonction float pour effectuer la conversion
# explicite en float, car la valeur saisie n'est pas forcément entière
largeur = input("Largeur de la pièce (m) : ")
```

```
longueur = input("Longueur de la pièce (m) : ")
surface = float(largeur) * float(longueur)
print(f"La surface de la pièce est : {surface} m²")
```

```
Largeur de la pièce (m) : 12
Longueur de la pièce (m) : 4.5
La surface de la pièce est : 54.0 m²
```

## 1.7 Les commentaires dans un programme

Le langage Python permet au programmeur de placer des commentaires dans son code : des commentaires sont des lignes de texte qui ne seront pas interprétées lors de l'exécution du programme. En Python, une ligne est en commentaire si elle commence par le caractère dièse. Les lignes suivantes sont en commentaires :

```
# Voici une ligne en commentaire
# print("si on décommente cette ligne, je m'affiche à l'exécution")
# On peut aussi mettre un commentaire sur la même ligne :
print("Bonjour") # Ici on est poli
```

Pour écrire un commentaire sur plusieurs lignes, on va utiliser cette notation:

```
"""
Ceci est
Un commentaire
Sur plusieurs lignes
"""
```

- Un des moyen d'analyser du code est d'en commenter l'ensemble des lignes, puis de décommenter une à une les lignes pour en observer le fonctionnement. A coupler avec l'utilisation de la fonction *print* pour afficher dans la console les résultats intermédiaires.
- Il est indispensable de commenter son code:
  - Description générale de ce que fait le programme.
  - Commentaires internes au code.
- Cependant les commentaires doivent rester pertinents.

```
# Commentaire non pertinent
# Initialisation de la variable price
prix = 100

# Commentaire pertinent
# Un nombre premier est un entier naturel qui admet exactement deux
# diviseurs distincts entiers et positifs (qui sont alors 1 et lui-même)
# ...
```

- Un code trop commenté est un code douteux (on fait trop compliqué).
- Il faut donc trouver le juste équilibre.

## 1.8 Pour aller plus loin

### 1.8.1 Python en ligne de commande

**La console Python** Pour des raisons de confort, nous avons utilisé Spyder pour écrire et exécuter un programme. Cependant cela n'est pas obligatoire, on peut très bien utiliser la console Python :  
- Sous Windows, taper *cmd* après avoir ouvert le menu de démarrage - Sous Linux / Mac, ouvrir un terminal Taper ensuite la commande :

```
python
```

Si l'installation de Python s'est effectuée correctement, de nouvelles lignes apparaissent dans l'invite de commande, par exemple :

```
Python 3.6.5 |Anaconda 4.1.1 (64-bit)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

La première commande que nous allons exécuter à l'aide de Python est très simple et consiste à afficher une chaîne de caractères. Rentrer la commande :

```
[3]: print("Hello World!")
```

```
Hello World!
```

Cette façon de faire peut être utile si l'on doit (ou si l'on veut) effectuer des opérations simples ou des tests. A noter que la console Python sert aussi de calculatrice, on peut par exemple effectuer le calcul suivant :

```
[2]: (2 + (3 * (10.5 / 4)) ** 2) / 11
```

```
[2]: 5.8196022727272725
```

### Exécuter un programme Python

A l'aide de l'invite de commande, se déplacer dans le répertoire contenant un programme écrit précédemment avec Spyder (indication : utiliser la commande "cd"). Ensuite exécuter la commande : `python mon_programme.py`.



# 20200409-biost1\_cent1\_est1\_synthese\_portee\_variables

April 9, 2020

## 1 Explication détaillée de la gestion des variables en Python

- En Python, le nom d'une variable peut être vu comme une étiquette pointant vers une zone mémoire. On peut récupérer l'identifiant d'une zone mémoire en appelant la fonction `id(nom_variable)`.
- Il existe deux grandes familles de variables :
  - Celles de type "non mutables" : il s'agit des types de base (`str`, `int`, `float`, `bool`) plus quelques autres (`tuples`, `frozen set`...)
  - Celles de type "mutables" : concerne les types "complexes" (`list`, `dict`, `set`, ...) qui peuvent être modifiés intrinsèquement (par l'indice, la clé...).

### 1.1 Portée des variables

L'endroit où est déclarée une variable détermine son niveau d'accessibilité dans le programme, c'est ce qu'on appelle la portée.

### 1.2 Variables non mutables

Si on affecte une valeur à une variable, puis qu'on affecte une autre valeur à cette même variable, on va créer deux zones mémoire distincte et le nom de la variable (l'étiquette) va pointer d'abord vers la première zone puis ensuite vers la seconde zone (la fonction `id` nous permet ici d'identifier la zone mémoire).

Exemple :

```
a = 2
print(id(a))  # ex. 139902503606208
a = 3
print(id(a))  # ex. 139902503606240
```

A noter que si le contenu est identique, l'id reste le même :

```
a = 2
print(id(a))  # ex. 139902503606208
a = 2
print(id(a))  # c'est le même
```

Notons aussi le comportement avec deux noms de variable différents pour un même contenu :

```
a = 2
print(id(a))  # ex. 139902503606208
b = a
print(id(b))  # c'est le même
```

### 1.3 Variables mutables

On les appelle variables mutables car on peut les modifier intrinsèquement sans changer de zone mémoire. Par exemple :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398225728
lst[0] = 8
print(id(lst))  # C'est le même
lst.append(20)
print(id(lst))  # C'est le même
```

En revanche, comme pour les variables non mutables, une nouvelle affectation provoque un changement d'id :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398225088
lst = [7, 2, 4]
print(id(lst))  # ex. 139902398225536
```

A noter que si le contenu est identique, l'id est tout de même changé (à la différence des variables non mutables) :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398224960
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398225408
```

Cependant comme pour les variables non mutables, deux noms de variables peuvent pointer vers le même contenu :

```
lst = [1, 2, 3]
print(id(lst))  # ex. 139902398224576
lst2 = lst
print(id(lst2))  # C'est le même
```

Corrolaire :

```
lst = [1, 2, 3]
lst2 = lst  # Ce n'est pas une copie, mais un "pointage" identique
lst2[0] = 6
print(lst2)
print(lst)  # lst a aussi été modifié
```



## 1.4 Lecture du contenu d'une variable en fonction de sa portée

### 1.4.1 Déclaration hors fonction : variable globale

Lorsqu'une variable est déclarée en dehors de toute fonction, c'est une **variable globale**, c'est à dire qu'elle est accessible (on peut lire son contenu) dans l'intégralité du programme.

```
def ma_fonction1():  
    # La variable est accessible  
    print(var_globale)  
  
# Programme principal  
var_globale = 3  
ma_fonction1()
```

### 1.4.2 Déclaration dans une fonction : variable locale

Lorsqu'une variable est déclarée dans une fonction (ou figurant en paramètre de la fonction), la variable n'est accessible que dans la fonction, c'est ce qu'on appelle une **variable locale**.

```
def ma_fonction2():  
    # La variable var_f2 est accessible uniquement dans la fonction  
    var_f2 = 1  
    print(var_f2)  
  
def ma_fonction3(mon_param):  
    # La variable mon_param est accessible uniquement dans la fonction  
    print(mon_param)  
  
# Programme principal  
ma_fonction2()  
print(var_f2) # Erreur, variable inconnue !  
  
ma_fonction3(4)  
print(mon_param) # Erreur, variable inconnue !
```

## 1.5 Modification du contenu d'une variable en fonction de sa portée

### 1.5.1 Variables de type non mutables (immuables)

Pour rappel, ce sont les variables ayant un type de base Python (str, int, float, bool) ainsi que quelques autres (tuples, frozen set...). Elles ne peuvent être "modifiées" qu'au niveau où elles ont été déclarées. Notons que le terme "modifiées" est quelque peu abusif, car comme nous l'avons vu précédemment, l'affectation d'une nouvelle valeur provoque le stockage dans une nouvelle zone mémoire et l'étiquette (le nom) de la variable pointera vers cette nouvelle zone.

- Une variable globale n'est modifiable que dans une zone hors fonction (dans le "programme principal"). Notons qu'il est tout de même possible de contourner cette limitation en spécifiant le mot-clé `global` suivi du nom de la variable globale dans une fonction. Cela est toutefois déconseillé car cela nuit à la lisibilité du programme.

- Une variable locale à une fonction n'est modifiable que dans cette fonction (c'est logique puisqu'elle n'est accessible en lecture que dans cette fonction).

```
def ma_fonction4():
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée ma_var, différente de la variable globale
    ma_var = 3

## PP
ma_var = 4
ma_fonction4()
print(ma_var)  # Le résultat est 4 !
```

Autre exemple avec un comportement similaire :

```
def ma_fonction4b(ma_var):
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée ma_var, différente de la variable passée en paramètre
    # (qui est en fait la variable globale)
    ma_var = 3

## PP
ma_var = 4
ma_fonction4b(ma_var)
print(ma_var)  # Le résultat est 4 !
```

### 1.5.2 Variables de type mutables

Pour rappel, il s'agit d'objets Python plus complexes (listes, dictionnaires, sets...).

- Une variable globale est non seulement accessible dans la fonction, mais modifiable (de façon intrinsèque) directement par cette dernière.
- Une variable locale à une fonction n'est accessible et modifiable que dans cette fonction (idem cas précédents).

```
def ma_fonction5():
    g_lst1.append(8)

def ma_fonction6(param_lst):
    param_lst.append(3)  # On modifie la liste initiale

## PP
g_lst1 = [1, 2, 3]
ma_fonction5()
print(g_lst1)  # [1, 2, 3, 8]

g_lst2 = [1]
ma_fonction6(g_lst2)
print(g_lst2)  # [1, 3]
```

- Corrolaire 1 : attention aux modifications des variables de type mutables dans les fonctions, ils le seront après l'appel de la fonction.
- Corrolaire 2 : si on souhaite qu'une fonction modifie une variable mutable définie au niveau global, il n'est pas nécessaire de la passer en paramètre (cf. la fonction `ma_fonction5`).

**Remarque:** si on réaffecte la variable mutable, alors on se retrouve comme dans le cas des variables immuables.

```
def ma_fonction6(param_lst):
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée param_lst mais différente de celle passée en paramètre
    param_lst = []

## PP
g_lst1 = [1, 2, 3]
ma_fonction6()
print(g_lst1)  # [1, 2, 3]
```

Même comportement ici :

```
def ma_fonction6():
    # L'instruction ci-dessous crée une nouvelle variable
    # également nommée g_lst1 mais différente de celle qui est globale
    g_lst1 = []

## PP
g_lst1 = [1, 2, 3]
ma_fonction6()
print(g_lst1)  # [1, 2, 3]
```



# 02-cours\_python\_conditions\_indent

January 11, 2020

## 1 Programmation conditionnelle et indentation

### 1.1 Condition

Une condition est le résultat d'une **évaluation** booléenne.

Voici les comparateurs :

- Egalité : `==`
- Non égalité (différent) : `!=`
- Supérieur (ou égal) : `>`, `>=`
- Inférieur (ou égal) : `<`, `<=`

```
[1]: nombre = 7
# Notez le comparateur avec un *double* signe '=' (qui n'est donc pas ici une ↵
↵ affectation !).
print(nombre == 7)
print(nombre != 7)
```

True

False

### 1.2 Conditions *if* / *elif* / *else*

```
[6]: # Exemple simple avec l'instruction if

nombre = 0
# Suite du programme
# ...
if nombre == 0: # nombre == 0 est une expression booléenne
    print("Bonjour, c'est votre jour de chance !")
```

Bonjour, c'est votre jour de chance !

```
[1]: # Exemple avec if / elif / else

valeur1 = 12
valeur2 = 14
```

```

if valeur1 > valeur2:
    print("valeur1 est plus grande que valeur2")
elif valeur1 == valeur2:
    print("valeur1 et valeur2 sont égales")
else:
    print("valeur1 est plus petite que valeur2")

```

valeur1 est plus petite que valeur2

## 1.3 Le rôle fondamental de l’indentation

### 1.3.1 Définition

- L’indentation définit un bloc d’instructions.
- L’indentation est obtenue en appuyant sur la touche “TAB” du clavier.

Par exemple ici les trois instructions *print* ne sont exécutées que si a vaut 2 :

```

# Notez la présence du caractère ":"
# Il marque le début d'un bloc
if a == 2:
    print(f"a vaut {a}")
    print("Ce programme est fascinant")
    print("Il ne peut être compris que par des êtres intelligents")

```

Remarques :

- Il ne faut jamais mixer les espaces et les tabulations, sinon les ennuis sont garantis.
- On peut régler son éditeur pour que lorsqu’on appuie sur la touche *tab* il génère 4 espaces.
- Le faire dans Spyder “Préférences... Editeur...”.

## 1.4 Exercices

### 1.4.1 Devinette

Sans taper le code, indiquer quelle est la différence entre ces deux extraits de code :

```

if a == 2:
    print("Je me surpasse en programmation")
    print("Je suis un Dieu en Python")

```

et :

```

if a == 2:
    print("Je me surpasse en programmation")
print("Je suis un Dieu en Python")

```

### 1.4.2 Exercice : VRAI / FAUX

Demander à l’utilisateur d’entrer un nombre décimal. À l’aide de Python, vérifier si le nombre rentré par l’utilisateur est égal à 11.5. Si oui, afficher le message “VRAI”, sinon afficher “FAUX”.

```
[3]: reponse = input("Entrez une valeur décimale : ")
      valeur_d = float(reponse)
      if valeur_d == 11.5:
          print("VRAI")
      else:
          print("FAUX")
```

```
Entrez une valeur décimale : 11.5
VRAI
```

### 1.4.3 Exercice : nombre pair ou impair

Demander à l'utilisateur de saisir un entier. Le programme affichera si le nombre est pair ou impair.

```
[5]: reponse = input("Saisissez un nombre entier : ")
      nb = int(reponse)
      if nb%2 == 0:
          print("Ce nombre est pair")
      else:
          print("Ce nombre est impair")
```

```
Saisissez un nombre entier : 11
Ce nombre est impair
```

### 1.4.4 Exercice : comparaisons

Afin de mettre en place un système de caisses automatiques, le directeur du cinéma "Les Studios" vous demande de coder un programme demandant au client son âge, s'il est abonné et lui annonçant en retour le tarif appliqué :

- Moins de 18 ans : 4 €
- Abonné : 5,50 €
- Sinon : 8 €

```
[6]: reponse = input("Quel est votre âge ? ")
      age = int(reponse)
      if age < 18:
          print("Tarif enfants : 4 €")
      else:
          abonne = input("Etes-vous abonné ? (o/n) ")
          if abonne == 'o':
              print("Tarif réduit : 5.50 €")
          else:
              print("Plein tarif : 8 €")
```

```
Quel est votre âge ? 19
Etes-vous abonné ? (o/n) o
Tarif réduit : 5.50 €
```

### 1.4.5 Exercice : le plus grand des nombres

Soit trois nombres,  $n_1$ ,  $n_2$  et  $n_3$ . Ecrire un programme indiquant lequel de ces trois nombres est le plus grand.

On peut utiliser le **module** *random* de Python pour “tirer” des nombres au hasard. Il s'utilise de la façon suivante :

```
[2]: import random

n1 = random.randint(0, 100)
print(n1)
```

41

```
[10]: import random

NB_MIN = 0  # Par convention, en majuscules, car c'est une constante
NB_MAX = 100

n1 = random.randint(NB_MIN, NB_MAX)
n2 = random.randint(NB_MIN, NB_MAX)
n3 = random.randint(NB_MIN, NB_MAX)

# On recherche le plus grand des 3
if n1 >= n2 and n1 >= n3 :
    plus_grand = n1
elif n2 >= n1 and n2 >= n3 :
    plus_grand = n2
else :
    plus_grand = n3

print(f"Nombres tirés : {n1}, {n2}, {n3}")
print(f"Nombre le plus grand : {plus_grand}")
```

Nombre tirés : 74, 14, 66

Nombre le plus grand 74

## 1.5 Le type booléen

Nous avons vu jusqu'à présent essentiellement trois types : les nombres entiers (*integer* en anglais, *int* en Python), les nombres décimaux (*float* en Python) et les chaînes de caractères (*string* en anglais, *str* en python). Nous avons pu manipuler un autre type: le type booléen (*boolean* en anglais, *bool* en Python). Ce type accepte deux valeurs : *True* et *False* (ce sont des mots-clés Python et ils commencent bien par une majuscule).



**1.5.1 Exercice : sans exécuter le programme, indiquer ce qu’affichent les instructions *print* ci-dessous**

```
a = 1
b = 2
c = 2
print(a == b)
print(b == c)
d = a == b
print(d)
print(type(d))
e = False # Attention à bien mettre la majuscule (idem pour True)
print(e)
f = True
if not d: # Équivalent à if d == False
    print("Pas bon!")
elif f:
    print("Toujours pas")
if f: # Équivalent à if f == True
    print("C'est tout bon !")
```

Notez que le choix du nom des variables est tout sauf explicite. Comme il s’agit d’un exercice de gymnastique mentale, cela est autorisé...

```
[5]: a = 1
      b = 2
      c = 2
      print(a == b)
      print(b == c)
      d = a == b
      print(d)
      print(type(d))
      e = False # Attention à bien mettre la majuscule (idem pour True)
      print(e)
      f = True
      if not d: # Équivalent à if d == False
          print("Pas bon!")
      elif f:
          print("Toujours pas")
      if f: # Équivalent à if f == True
          print("C'est tout bon !")
```

```
False
True
False
<class 'bool'>
False
Pas bon!
C'est tout bon !
```



# 03-cours\_python\_boucle\_while

January 11, 2020

## 1 Les boucles *while*

### 1.1 Généralités

En informatique, les boucles sont des instructions qui permettent de répéter des séries d'opérations un certain nombre de fois (voire même un nombre infini de fois). Les boucles sont donc très utiles pour effectuer des opérations répétitives. En Python, il existe deux types de boucles: ***while*** et ***for***. Au cours de cette séance, nous allons étudier le premier type: les boucles ***while***.

L'instruction *while* s'utilise de la façon suivante :

```
while condition_booleene:
    # Instructions si la condition est vérifiée
    # ...
```

**Exemple :** l'instruction suivante permet d'afficher les nombres entiers compris entre 0 et 6 inclus :

```
[1]: compteur = 0
# Ce qui suit l'instruction while est une *condition booléenne*
while compteur <= 6:
    print(compteur)
    compteur += 1 # équivalent à compteur = compteur + 1
print("Fin boucle !")
print(compteur)
```

```
0
1
2
3
4
5
6
Fin boucle !
7
```

### 1.2 L'instruction *break*

On peut également créer une boucle dont la condition est toujours vraie et sortir "brutalement" avec l'instruction ***break*** :

```
[2]: compteur = 0
while True: # Ceci est toujours vrai
    if compteur > 6:
        break # On sort de la boucle
    compteur += 1
print(compteur)
```

7

Il n'est pas rare de voir cette instruction *while True* dans des programmes Python. Attention cependant aux boucles infinies, c'est à dire qui font que l'instruction *break* n'est jamais exécutée et donc qu'on ne sort jamais de la boucle...

### 1.3 Exercices

#### 1.3.1 Exercice : valeurs décroissantes

Demander à l'utilisateur une valeur entière. Afficher la liste des valeurs entières qui suivent dans l'ordre décroissant, tant qu'on n'a pas atteint la valeur 0.

```
[3]: valeur = int(input("valeur : "))
i = valeur
while i > 0:
    print(i)
    i = i - 1 # ou i -= 1
```

valeur : 5

5  
4  
3  
2  
1

#### 1.3.2 Exercice : affichage d'une suite de nombres

Ecrire un programme utilisant une boucle *while* de telle sorte qu'on ait les valeurs : 1, 2, 4, 8, 16, 32, 64, 128. On affichera les nombre les uns en dessous des autres.

```
[4]: n = 7
i = 0
while i <= n:
    print(2**i)
    i += 1
```

1  
2  
4  
8  
16  
32

64  
128

### 1.3.3 Exercice : que fait le programme suivant ?

```
i = 1
while True:
    if i > 7:
        print("Je sors !")
```

Euh... il fait chaud...

### 1.3.4 Exercice : concaténation de chaînes de caractères dans une boucle

- Demander à l'utilisateur un nombre
- Afficher dans une chaîne toutes les valeurs de 1 jusqu'à ce nombre

Exemple :

```
n = 4
# Résultat
"1 2 3 4"
```

```
[5]: # Version 1
n = int(input("n : "))
i = 1
resultat = ""
while i <= n:
    resultat = resultat + " " + str(i)
    i = i+1
print(resultat)
```

```
n : 5
1 2 3 4 5
```

```
[6]: # Version 2
n = int(input("n : "))
i = 1
resultat = ""
while i <= n:
    resultat = f"{resultat} {i}"
    i = i+1
print(resultat)
```

```
n : 5
1 2 3 4 5
```

Que remarque t-on sur l'affichage ?

## 1.4 Exercice : boucler tant qu'on n'a pas la bonne réponse

- Demander à l'utilisateur de répondre *oui* ou *non*.
- Si la réponse n'est pas correcte afficher un message d'erreur et reposer la question.
- S'il a répondu *oui* ou *non*, afficher *Bravo !*.

```
[1]: # 1ère version

reponse = input("Répondre par oui ou par non : ")
while reponse != "oui" and reponse != "non":
    print("Reponse incorrecte... merci d'être attentif !")
    reponse = input("Répondre par oui ou par non : ")
print("Bravo !")
```

```
Répondre par oui ou par non : hello
Reponse incorrecte... merci d'être attentif !
Répondre par oui ou par non : oui
Bravo !
```

```
[ ]: # 2ème version

while True:
    reponse = input("Répondre par oui ou par non : ")
    if reponse == "oui" or reponse == "non":
        break
    else: # else est superflu ici, pourquoi ?
        print("Reponse incorrecte... merci d'être attentif !")
print("Bravo !")
```

### 1.4.1 Exercice : boucle while avec condition d'arrêt - deviner un nombre aléatoire :

- Le programme tire un nombre aléatoire
- On demande à l'utilisateur de saisir un nombre
- On lui indique si il est trop petit, trop grand ou s'il a gagné
- Lorsqu'il a gagné on indique à l'utilisateur en combien de fois

```
[ ]: import random

# On utilise ci-dessous les majuscules pour indiquer que ce sont des constantes
# Ce n'est qu'une convention... (PEP8)
MIN = 1
MAX = 20

nb_saisies = 0
valeur_a_trouver = random.randint(MIN, MAX)
while True:
    valeur_saisie = int(input(f"Saisissez une valeur entre {MIN} et {MAX} "))
    nb_saisies += 1
```

```

if valeur_saisie < valeur_a_trouver:
    print("Valeur trop petite")
elif valeur_saisie > valeur_a_trouver:
    print("Valeur trop grande")
else:
    print("Bravo, c'est gagné !")
    break

print(f"Vous avez deviné en {nb_saisies} fois.")

```

#### 1.4.2 Exercice : tracer un trait avec l'utilisation de la bibliothèque turtle

Tracer un trait de 100 pixels de façon saccadée, de 10 pixels en 10 pixels.

- On utilise le module turtle en écrivant: `import turtle` (as tu).
- [Documentation \(non exhaustive\) et exemples](#).
- Pour marquer des pauses entre le traçage des traits, on utilisera la fonction `sleep(nb_secondes)` du module `time`.

```

import time
import turtle as tu

turtle1 = tu.Turtle()
while ...:
    ...
tu.done()

```

```

[1]: # Solution 1

import time
import turtle as tu

my_turtle = tu.Turtle()
my_turtle.goto(0, 0)
position = 0
while position < 100:
    my_turtle.forward(10)
    time.sleep(0.5)
    position += 10
tu.done()

```

```

↳ -----
Terminator                                Traceback (most recent call↳
↳last)

```

```

<ipython-input-1-0469a4164872> in <module>

```

```
11     time.sleep(0.5)
12     position += 10
--> 13 tu.done()
```

/usr/lib/python3.7/turtle.py in mainloop()

Terminator:

```
[ ]: # Solution 2 (plus élégante)

import time
import turtle as tu

my_turtle = tu.Turtle()
my_turtle.goto(0, 0)
while my_turtle.position()[0] < 100:
    my_turtle.forward(10)
    time.sleep(0.5)

tu.done()
```



# 04-cours\_python\_listes\_boucles

January 17, 2020

## 1 Les listes, les boucles

Une liste est une suite **ordonnée** d'objets (entiers, chaînes, décimaux...) sur lesquels on va pouvoir effectuer toute une série d'opérations (ajout, modification, suppression, extraction...).

### 1.1 Initialisation / construction des listes

```
[ ]: liste_nombres = [1, 2, 3]
      liste_chaines_1 = ["1", "2", "3"]
      liste_chaines_2 = ["bonjour", "hello", "python", "ISEN"]
      liste_melangee = [29, "a", 22, 3.14, 35, "b", "ISEN Bretagne", 56, 44]
      liste_de_listes = [[1, 2], [3, 4]] # Liste contenant des listes
      liste_vide = [] # Liste vide qu'on va pouvoir ensuite construire dynamiquement
```

### 1.2 Extraction d'éléments d'une liste

#### 1.2.1 Indicage

Pour l'extraction d'éléments contenus dans une liste, on peut utiliser l'indicage, qui consiste à donner la position de l'élément dans la liste. Le premier élément d'une liste commence à **0**.

#### 1.2.2 Exercice

Etant données les listes suivantes :

```
liste_melangee = [29, "a", 22, 3.14, 35, "b", "ISEN Bretagne", 56, 44]
liste_de_listes = [[1, 2], [3, 4]]
```

Indiquer le contenu des éléments suivants :

```
liste_melangee[0]
liste_melangee[3]
liste_melangee[8]
liste_de_listes[0]
liste_de_listes[1]
liste_de_listes[0][0]
liste_de_listes[1][0]
liste_melangee[-1]
liste_melangee[-2]
liste_melangee[9]
```

```
[1]: liste_melangee = [29, "a", 22, 3.14, 35, "b", "ISEN Bretagne", 56, 44]
    liste_de_listes = [[1, 2], [3, 4]]

    print(liste_melangee[0])
    print(liste_melangee[3])
    print(liste_melangee[8])
    print(liste_de_listes[0])
    print(liste_de_listes[1])
    print(liste_de_listes[0][0])
    print(liste_de_listes[1][0])
    print(liste_melangee[-1])
    print(liste_melangee[-2])
    print(liste_melangee[9])
```

```
29
3.14
44
[1, 2]
[3, 4]
1
3
44
56
```

```

↳
↳ -----
↳
↳ IndexError                                Traceback (most recent call↳
↳ last)
↳
↳ <ipython-input-1-327f8fd76904> in <module>
↳     11 print(liste_melangee[-1])
↳     12 print(liste_melangee[-2])
↳ ---> 13 print(liste_melangee[9])
```

```
IndexError: list index out of range
```

### 1.3 Extraction d'une sous-liste

Pour extraire des sous-listes, on procède de la façon suivante :

```
sous_liste = liste[m:n]
```

**Attention !** : la sous-liste contiendra les éléments de la liste originale depuis l'indice  $m$  **inclus** jusqu'à l'indice  $n$  **exclu** (en d'autres termes, de l'indice  $m$  à l'indice  $n-1$  tous deux inclus).

- Si l'indice  $n$  n'est pas précisé, on commence à l'indice  $m$ .

- Si l'indice  $m$  n'est pas précisé, on va jusqu'à l'indice  $n$  exclu.

Quel est le contenu des éléments suivants ?

```
liste_melangee = [29, 'a', 22, 3.14, 35, 'b', "ISEN Bretagne", 56, 44]
liste_melangee[0:2]
liste_melangee[6:9]
liste_melangee[3:4]
liste_melangee[3]
liste_melangee[2:]
liste_melangee[:3]
liste_melangee[:-1]
```

```
[2]: liste_melangee = [29, 'a', 22, 3.14, 35, 'b', "ISEN Bretagne", 56, 44]
print(liste_melangee[0:2])
print(liste_melangee[6:9])
print(liste_melangee[3:4])
print(liste_melangee[3])
print(liste_melangee[2:])
print(liste_melangee[:3])
print(liste_melangee[:-1])
```

```
[29, 'a']
['ISEN Bretagne', 56, 44]
[3.14]
3.14
[22, 3.14, 35, 'b', 'ISEN Bretagne', 56, 44]
[29, 'a', 22]
[29, 'a', 22, 3.14, 35, 'b', 'ISEN Bretagne', 56]
```

## 1.4 La fonction len

La fonction **len** permet de récupérer la longueur d'une liste.

```
[5]: liste = [1, 4, 12, "hello"]
longueur = len(liste)
print(longueur)
```

4

### 1.4.1 Remarque : quel est résultat affiché par le code suivant ?

```
liste = ["a", 1, [1, 2, 3]]
print(len(liste))
```

### 1.4.2 Exercice : somme avec une boucle while

Soit la liste : [404, 8, 22, 310, 5]. A l'aide d'une boucle *while* effectuer la somme des éléments de cette liste et l'afficher à la fin.

```
[4]: liste = [404, 8, 22, 310, 5]
    somme = 0
    i = 0
    while i < len(liste):
        somme = somme + liste[i]
        i = i + 1
    print(somme)
```

749

### 1.4.3 Exercice : calcul de la valeur minimale d'une liste

Soit la liste: `liste = [1, 5, 12, 2, 27]`. En utilisant une boucle *while*, écrire un programme qui trouve la valeur minimale de cette liste et l'affiche à la fin. Avant d'écrire le programme Python, il est recommandé de réfléchir sur papier à un algorithme permettant la résolution du problème.

```
[6]: liste = [31, 5, 12, 2, 27]
    val_min = liste[0]
    i = 1
    while i < len(liste):
        if liste[i] < val_min:
            val_min = liste[i]
        i += 1
    print(val_min)
```

2

### 1.4.4 Exercice : calcul du montant de la liste des courses

On dispose d'une liste de produits et de prix :

```
liste_produits = ['tomates', 'gâteaux apéro', 'bière']
liste_prix = [3, 2.5, 4.6]
```

- L'utilisateur peut choisir un produit
- On ajoute le prix correspondant dans un montant total
- On repète cela tant que l'utilisateur ne tape pas "quitter"
- A la fin on lui affiche le montant total

```
[7]: # Version 1
    liste_produits = ['tomates', 'gâteaux apéro', 'bière']
    liste_prix = [3, 2.5, 4.6]

    reponse = ""
    montant_courses = 0
    while reponse != "quitter":
        reponse = input("No. de votre produit (ou quitter) : ")
        if reponse != "quitter":
            num_produit = int(reponse)
            print(f"Vous avez choisi :{liste_produits[num_produit]}")
```

```

montant_courses += liste_prix[num_produit]
print(f"Fin des courses, total : {montant_courses} Euros")

```

```

No. de votre produit (ou quitter) : 1
Vous avez choisi :gâteaux apéro
No. de votre produit (ou quitter) : 2
Vous avez choisi :bière
No. de votre produit (ou quitter) : quitter
Fin des courses, total : 7.1 Euros

```

```

[7]: # Version 2
liste_produits = ['tomates', 'gâteaux apéro', 'bière']
liste_prix = [3, 2.5, 4.6]

montant_courses = 0
while True:
    reponse = input("No. de votre produit (ou quitter) : ")
    if reponse != "quitter":
        num_produit = int(reponse)
        print(f"Vous avez choisi :{liste_produits[num_produit]}")
        montant_courses += liste_prix[num_produit]
    else:
        break
print(f"Fin des courses, total : {montant_courses} Euros")

```

```

No. de votre produit (ou quitter) : 1
Vous avez choisi :gâteaux apéro
No. de votre produit (ou quitter) : 2
Vous avez choisi :bière
No. de votre produit (ou quitter) : quitter
Fin des courses, total : 7.1 Euros

```

```

[6]: # Version 3
liste_produits = ['tomates', 'gâteaux apéro', 'bière']
liste_prix = [3, 2.5, 4.6]

montant_courses = 0
reponse = input("No. de votre produit (ou quitter) : ")
while reponse != "quitter":
    num_produit = int(reponse)
    print(f"Vous avez choisi :{liste_produits[num_produit]}")
    montant_courses += liste_prix[num_produit]
    reponse = input("No. de votre produit (ou quitter) : ")

print(f"Fin des courses, total : {montant_courses} Euros")

```

```

No. de votre produit (ou quitter) : 1
Vous avez choisi :gâteaux apéro

```

```
No. de votre produit (ou quitter) : 2
Vous avez choisi :bière
No. de votre produit (ou quitter) : quitter
Fin des courses, total :7.1 Euros
```

#### 1.4.5 Exercice : calcul du montant de la liste des courses (version sécurisée)

Comme il n'est pas raisonnable de vendre de l'alcool aux personnes ayant moins de 18 ans, on commencera par demander l'âge au client. Ainsi au moment du choix d'un produit, s'il est alcoolisé et si le client est mineur, on refusera l'achat.

Pour indiquer si un produit contient de l'alcool, on pourra utiliser une 3ème liste :

```
liste_produits = ['tomates', 'gâteaux apéro', 'bière']
liste_prix = [3, 2.5, 4.6]
liste_alcools = [False, False, True]
```

**Cas d'usage :** pour vérifier si le programme fonctionne, penser à tester les cas suivants :

- Age < 18 et choix alcool, le choix est refusé et le produit ne doit pas être ajouté au total
- Age >= 18 et choix alcool, le choix est accepté et le produit doit être ajouté au total
- Age < 18 et choix non alcool, le choix est accepté et le produit doit être ajouté au total
- Age >= 18 et choix non alcool, le choix est accepté et le produit doit être ajouté au total

```
[1]: liste_produits = ['tomates', 'gâteaux apéro', 'bière']
liste_prix = [3, 2.5, 4.6]
liste_alcools = [False, False, True]

reponse = ""
montant_courses = 0
age = int(input("Quel est votre âge ? "))
while reponse != "quitter":
    reponse = input("No. de votre produit (ou quitter) : ")
    if reponse != "quitter":
        num_produit = int(reponse)
        print(f"Vous avez choisi :{liste_produits[num_produit]}")
        if age < 18 and liste_alcools[num_produit]: # == True implicite
            print("Vous ne pouvez acheter de l'alcool !")
        else:
            montant_courses += liste_prix[num_produit]
print(f"Fin des courses, total : {montant_courses} Euros")
```

```
Quel est votre âge ?12
No. de votre produit (ou quitter) : 2
Vous avez choisi :bière
Vous ne pouvez acheter de l'alcool !
No. de votre produit (ou quitter) : 1
Vous avez choisi :gâteaux apéro
No. de votre produit (ou quitter) : quitter
Fin des courses, total : 2.5 Euros
```

#### 1.4.6 Exercice : calcul du montant de la liste des courses (avec promo)

On décide d'appliquer des promotions sur certains produits (en fait ici pour simplifier des coefficients). Modifier le programme de telle sorte que les promotions soient appliquées sur le prix. Pour ce faire, on utilisera une 4ème liste :

```
liste_produits = ['tomates', 'gâteaux apéro', 'bière']
liste_prix = [3, 2.5, 4.6]
liste_alcools = [False, False, True]
liste_promos = [1, 0.8, 0.7]
```

```
[2]: liste_produits = ['tomates', 'gâteaux apéro', 'bière']
liste_prix = [3, 2.5, 4.6]
liste_alcools = [False, False, True]
liste_promos = [1, 0.8, 0.7]

reponse = ""
montant_courses = 0
age = int(input("Quel est votre âge ? "))
while reponse != "quitter":
    reponse = input("No. de votre produit (ou quitter) : ")
    if reponse != "quitter":
        num_produit = int(reponse)
        print(f"Vous avez choisi :{liste_produits[num_produit]}")
        if age < num_produit and liste_alcools[num_produit]: # == True
            ↪ implicite
            print("Vous ne pouvez acheter de l'alcool !")
        else:
            montant_courses += liste_prix[num_produit] *
            ↪ liste_promos[num_produit]
print(f"Fin des courses, total : {round(montant_courses, 2)} Euros")
```

```
Quel est votre âge ? 19
No. de votre produit (ou quitter) : 2
Vous avez choisi :bière
No. de votre produit (ou quitter) : quitter
Fin des courses, total : 3.22 Euros
```

#### 1.4.7 Exercice : inversion de l'ordre des éléments d'une liste

- Soit la liste : liste\_mots = ["Python", "de", "cours", "le", "J'aime"].
- Inverser l'ordre des mots de façon à obtenir : liste\_mots = ["J'aime", "le", "cours", "de", "Python"].

```
[1]: # 1ère solution
liste_mots = ["Python", "de", "cours", "le", "J'aime"]
i = 0
j = len(liste_mots) - 1
while i != j:
```

```

    val_tmp = liste_mots[i]
    liste_mots[i] = liste_mots[j]
    liste_mots[j] = val_tmp
    i += 1
    j -= 1
print(liste_mots)

```

```
["J'aime", 'le', 'cours', 'de', 'Python']
```

```

[2]: # 2ème solution (sans la variable val_tmp)
liste_mots = ["Python", "de", "cours", "le", "J'aime"]
i = 0
j = len(liste_mots) - 1
while i != j:
    liste_mots[i], liste_mots[j] = liste_mots[j], liste_mots[i]
    i += 1
    j -= 1
print(liste_mots)

```

```
["J'aime", 'le', 'cours', 'de', 'Python']
```

## 1.5 Parcours des éléments d'une liste à l'aide de l'instruction *for*

La boucle **for** permet de parcourir facilement une suite d'éléments, en s'affranchissant de la gestion de l'index (la variable *i* dans les exemples précédents), nécessaire dans une boucle *while*.

Voici une version du calcul de la somme des éléments d'une liste avec une boucle *for* :

```

[2]: liste = [7, 8, 10, 3]
somme = 0
for elem in liste:
    print(f"Élément courant: {elem}")
    somme = somme + elem
print(f"Somme: {somme}")

```

```

Élément courant: 7
Élément courant: 8
Élément courant: 10
Élément courant: 3
Somme: 28

```

- *elem* est le nom d'une variable qui pointe sur l'élément courant de la liste parcourue. Son type est donc fonction de l'élément courant de la liste.
- On retrouve la même structure que pour la boucle *while*, à savoir l'indentation après la ligne **for...** : qui indique que **toutes** les instructions **identées** sous la ligne *for* seront exécutées à **chaque tour de boucle**.



### 1.5.1 Exercice : calcul de la valeur minimale d'une liste (v2)

Soit la liste: `liste = [1, 5, 12, 2, 27]`. Nous avons précédemment calculé la valeur minimale en utilisant une boucle `while`. Faire de même mais cette fois en utilisant une boucle `for`.

```
[4]: liste = [31, 5, 12, 2, 27]
    val_min = liste[0]
    for elem in liste:
        if elem < val_min:
            val_min = elem
    print(f"Valeur minimale: {val_min}")
```

Valeur minimale: 2

### 1.5.2 Exercice : somme conditionnelle

Ecrire un programme (et un seul) qui effectue la somme des nombres d'une liste, mais en excluant ceux compris entre les éléments 's' et 'e' inclus.

Exemples :

- [1, 2, 2, 's', 99, 99, 'e'] : afficher 5
- [1, 1, 's', 'e', 2] : afficher 4
- [1, 1, 's', 3, 'e', 2] : afficher 4
- [1, 1, 's', 'e', 2] : afficher 4
- [1, 2, 2] : afficher 5
- On part de l'hypothèse que s'il y a un 's' il y a forcément un 'e' plus loin.
- Vérifier que votre programme fonctionne pour chacun des éléments ci-dessus (au moins).

```
[17]: liste = [1, 2, 's', 99, 99, 'e', 8]
    somme = 0
    faire_somme = True
    for elem in liste:
        if elem == 's':
            faire_somme = False
        elif elem == 'e':
            faire_somme = True
        elif faire_somme:
            somme = somme + elem
    print(somme)

# Remarque : on pourrait réaliser la même chose avec 3 boucles while à la suite
```

## 1.6 Le mot clé *in*

Outre son utilisation dans une boucle *for* comme vu précédemment, on peut utiliser l'instruction *in* pour savoir si un élément appartient à une liste.

Effectuer les tests suivants :

```
1 in [3, 1, 4]
reponse = "oui"
reponse in ["oui", "non"]
```

Quelle est le type de la valeur renvoyée par l'expression *in* ?

### 1.6.1 Exercice : recherche d'une valeur dans une liste

Soit la liste suivante : [12, 4, 5, 7]. Demander à l'utilisateur de saisir un nombre. En fonction de sa réponse, lui indiquer si ce nombre appartient à la liste.

```
[8]: liste = [12, 4, 5, 7]
nombre = int(input("Votre nombre ? "))
if nombre in liste:
    print("Il est dans la liste")
else:
    print("Il ne se trouve pas dans la liste")
```

Votre nombre ? 4

Il est dans la liste

## 1.7 La fonction *range()* et la boucle *for*

Pour boucler *n* fois avec la boucle *for*, on peut utiliser la fonction *range()*.

```
[2]: print("Exemple 1")
for i in range(0, 5): # Boucle 5 fois à partir de 0 et i commence à 0
    print(i)
# Remarque 1 : ceci est équivalent : for i in range(5):
# Remarque 2 : on peut effectuer la même chose avec une boucle while
#             mais c'est plus compliqué...

print("Exemple 2")
for i in range(2, 5): # Boucle 5 fois à partir de 0 et i commence à 2
    print(i)

print("Exemple 3")
for i in range(2, 5, 2): # Boucle 5 fois à partir de 0 et i commence à 2, par
    ↪ pas de 2
    print(i)
```

Exemple 1

0

1

```

2
3
4
Exemple 2
2
3
4
Exemple 3
2
4

```

### 1.7.1 Exercice : série mathématique

Ecrire un programme permettant le calcul de la série suivante :

$$S_n = \sum_{i=0}^n \left(\frac{1}{2}\right)^i$$

On fixera arbitrairement  $n$  en début de programme.

```

[1]: n = 25
      somme = 0
      for i in range(n+1):
          somme = somme + 0.5**i
      print(somme)

```

```
1.9999999701976776
```

## 1.8 Exercice : la suite de Fibonacci

Son inventeur est Léonard de Pise (1175 – v.1250), aussi connu sous le nom de Leonardo Fibonacci, qui a rapporté d'Orient la notation numérique indo-arabe et a écrit et traduit des livres influents de mathématiques.

Introduite comme problème récréatif dans son fameux ouvrage Liber Abaci, la suite de Fibonacci peut être considérée comme le tout premier modèle mathématique en dynamique des populations ! En effet, elle y décrit la croissance d'une population de lapins sous des hypothèses très simplifiées, à savoir : chaque couple de lapins, dès son troisième mois d'existence, engendre chaque mois un nouveau couple de lapins, et ce indéfiniment ([source](#)).

$$U_n = U_{n-1} + U_{n-2} \text{ avec } n \geq 2$$

Ecrire un programme qui demande à l'utilisateur la valeur  $n$  et qui affiche le résultat de la suite.

```

[4]: # Une première version...
      n = int(input("Nombre n : "))
      if n < 2:
          print("Merci de saisir un nombre supérieur ou égal à deux...")

```

```

else:
    u_n_1 = 1
    u_n_2 = 0
    u_n = u_n_1 + u_n_2
    for i in range(2, n):
        u_n_2 = u_n_1
        u_n_1 = u_n
        u_n = u_n_1 + u_n_2

print(u_n)

```

Nombre n : 4  
3

```

[ ]: # Une deuxième version
n = int(input("Nombre n : "))
if n < 2:
    print("Merci de saisir un nombre supérieur ou égal à deux...")
else:
    u1,u2 = 1, 1
    for i in range(1, n):
        u1,u2 = u2, u1+u2
    print(u1)

```

### 1.8.1 Exercice : affichage des nombres impairs de 1 à 10

- 1) Avec une boucle for et la fonction range()
- 2) Faire la même chose avec une boucle while

```

[8]: for i in range(1, 11, 2):
    print(i)

```

1  
3  
5  
7  
9

```

[7]: i = 1
while i < 11:
    if i%2 != 0:
        print(i)
    i += 1

```

1  
3  
5

7  
9

### 1.8.2 Exercice : affichage d'un triangle de '#'

Avec une boucle `for` et la fonction `range()`, afficher le “graphique” suivant :

```
#  
##  
###  
####  
#####  
#####
```

Indication : pour répéter `/n/` fois le même caractère on peut utiliser l'opérateur `*` sur un caractère.

Exemple : `"s" * 4 -> "ssss"`

```
[10]: for i in range(1, 7):  
       print("#" * i)
```

```
#  
##  
###  
####  
#####  
#####
```

## 1.9 Ajout d'éléments à une liste

L'ajout d'un élément s'effectue avec la méthode (~fonction) **`append`**.

```
[10]: liste = [29, "a", 22, 3.14, 35]  
      print(liste)  
      liste.append(123456789)  
      print(liste)
```

```
[29, 'a', 22, 3.14, 35]  
[29, 'a', 22, 3.14, 35, 123456789]
```

- Notons que l'utilisation de la méthode *append* est différente de *len*.
  - `len(liste)`
  - `liste.append(...)`
- Après appel à la méthode *append*, il y a directement **modification de la liste originale**.
- L'intérêt principal de la méthode *append* est la construction dynamique de liste. Ecrire le programme suivant et regarder ce qui se passe :

```
liste = []  
i = 0  
while i < 5:  
    liste.append(i)
```

```
print(liste)
i += 1
```

### 1.9.1 Exercice : concaténation de listes

Soient les listes suivantes :

```
liste1 = [1, 2, 3]
liste2 = [4, 5, 6, 7]
```

En utilisant l’instruction *for* et la méthode *append*, ajouter dans une troisième liste les éléments de *liste1* et *liste2*.

```
[18]: liste1 = [1, 2, 3]
      liste2 = [4, 5, 6, 7]
      liste3 = []
      for elem in liste1:
          liste3.append(elem)

      for elem in liste2:
          liste3.append(elem)

      print(liste3)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

### 1.9.2 Exercice : liste construite par l’utilisateur

- Demander à l’utilisateur de saisir une valeur (peu importe le type) et l’ajouter dans une liste. Continuer tant qu’il n’a pas saisi “-1”.
- Lorsqu’il a saisi “-1” afficher la liste ainsi que le nombre d’éléments.

```
[1]: valeur = ""
      liste = []
      while valeur != "-1": # Pourquoi y a t-il des guillemets ici ?
          valeur = input("Valeur ? (-1 pour terminer) : ")
          if valeur != "-1":
              liste.append(valeur)
      print(liste)
      print(len(liste))
```

```
Valeur ? (-1 pour terminer) : 12
Valeur ? (-1 pour terminer) : 3
Valeur ? (-1 pour terminer) : -1
['12', '3']
2
```

```
[ ]: valeur = ""
      liste = []
      while valeur != "-1": # Pourquoi y a t-il des guillemets ici ?
```

```

    valeur = input("Valeur ? (-1 pour terminer) : ")
    liste.append(valeur)

liste = liste[:-1]  # On supprime le dernier élément de la liste
print(liste)
print(len(liste))

```

### 1.9.3 Exercice : extraction des nombres impairs

Pour une liste donnée de nombres, extraire ceux qui sont impairs en les stockant dans une liste.

- Exemple: `liste = [1, 3, 4, 6, 7, 11, 12]` -> `[1, 3, 7, 11]`
- Indications : on utilisera une seconde liste qui contiendra uniquement les nombres impairs

```

[24]: liste = [1, 3, 4, 6, 7, 11, 12]
      liste_impairs = []
      for nombre in liste:
          if nombre % 2 != 0:
              liste_impairs.append(nombre)

      print(liste_impairs)

```

`[1, 3, 7, 11]`

### 1.10 Exercice : suppression des doublons

- Soit la liste : `[1, 2, 2, 3, 4, 2, 7, 1, 4, 5, 7, 6]`
- Créer une nouvelle liste en supprimant les éléments en double

```

[1]: liste_init = [1, 2, 2, 3, 4, 2, 7, 1, 4, 5, 7, 6]
      liste_res = []
      for elem in liste_init:
          if elem not in liste_res:
              liste_res.append(elem)
      print(liste_res)

```

`[1, 2, 3, 4, 7, 5, 6]`

### 1.11 Concaténation / extension de listes

La concaténation peut s'effectuer avec l'opérateur `+`.

```

[16]: liste1 = [1, 2, 3]
      liste2 = [4, 5, 6, 7]
      liste3 = liste1 + liste2
      print(liste3)

```

`[1, 2, 3, 4, 5, 6, 7]`

Si on souhaite étendre une liste existante, on peut utiliser la méthode **extend**.

```
[20]: liste1 = [1, 2, 3]
      liste2 = [4, 5, 6, 7]
      liste1.extend(liste2)
      print(liste2)
      print(liste1)
```

```
[4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
```

Que se passe t-il si on utilise la à la place d'*extend* on effectue : `liste1.append(liste2)` ?

## 1.12 Modification d'éléments d'une liste

Les éléments individuels d'une liste peuvent être modifiés (le type *list* est dit *mutable*). Tester l'exécution des lignes suivantes :

```
liste = [29, "a", 22, 3.14, 35, "b", "ISEN Bretagne", 56, 44]
liste[2] += 5
liste[6] = "Yncrea"
print(liste)
```

Que se passe t-il si on exécute : `liste[1] += 2` ?

### 1.12.1 Exercice : remplacement d'éléments (remplacement de mots)

Soit la liste : `["il", "fait", "beau", "ce", "soir"]`.

En utilisant une boucle *while*, remplacer les mots “beau” par “chaud” et “ce” par “le”.

```
[24]: liste = ["il", "fait", "beau", "ce", "soir"]
      i = 0
      lg_liste = len(liste)
      while i < lg_liste:
          if liste[i] == "beau":
              liste[i] = "chaud"
          elif liste[i] == "ce":
              liste[i] = "le"
          i += 1
      print(liste)
```

```
['il', 'fait', 'chaud', 'le', 'soir']
```

### 1.12.2 Exercice : remplacement d'éléments (remplacement de mots - v2)

Faire la même chose que l'exercice précédent, sachant que cette fois on a deux listes :

- `liste_originale = ["il", "fait", "beau", "ce", "soir"]`
- `liste_replacements = [None, None, "chaud", "le", None]`

`liste_originale` contiendra à la fin du programme : `['il', 'fait', 'chaud', 'le', 'soir']`



```
[2]: liste_originale = ["il", "fait", "beau", "ce", "soir"]
liste_replacements = [None, None, "chaud", "le", None]
i = 0
lg_liste = len(liste_originale)
while i < lg_liste:
    if liste_replacements[i] != None:
        liste_originale[i] = liste_replacements[i]
    i+=1

print(liste_originale)
```

['il', 'fait', 'chaud', 'le', 'soir']

Python “avancé” : plutôt que d’utiliser *while* et un indice (*i*), on peut utiliser la fonction **enumerate**. Ainsi le programme suivant est rigoureusement équivalent :

```
[ ]: liste = ["il", "fait", "beau", "ce", "soir"]
for i, element in enumerate(liste):
    if element == "beau": # ou: liste[i] == "beau"
        liste[i] = "chaud"
    elif element == "ce":
        liste[i] = "le"
print(liste)
```

A chaque tour de boucle, la variable *i* contient l’indice courant (en commençant par 0) et la variable *element* l’élément courant de la liste.

### 1.12.3 Exercice : modification des éléments d’une liste (valeurs au carré)

- Soit la liste suivante : `liste = [4, 5, 6]`
- Modifier cette liste de telle façon à remplacer chaque nombre par sa valeur au carré
- On devra donc obtenir : `[16, 25, 36]`

```
[28]: liste = [4, 5, 6]
i = 0
while i < len(liste):
    liste[i] = liste[i]**2
    i += 1
print(liste)
```

[16, 25, 36]

### 1.13 Suppression d’éléments dans une liste

On peut supprimer un élément de la liste en utilisant la méthode **remove(nom\_element)** ou **pop(indice\_element)**. Notons que de façon similaire à la méthode *append* vu précédemment, après appel à *remove* / *pop*, la liste originale est modifiée.

```
[26]: liste = [29, 'a', 22, 3.14, 35, 'b', "ISEN Bretagne", 56, 44]
print(liste)
liste.remove("ISEN Bretagne")
print(liste)
```

```
[29, 'a', 22, 3.14, 35, 'b', 'ISEN Bretagne', 56, 44]
[29, 'a', 22, 3.14, 35, 'b', 56, 44]
```

```
[1]: liste = [29, 'a', 22, 3.14, 35, 'b', "ISEN Bretagne", 56, 44]
print(liste)
liste.pop(6)
print(liste)
```

```
[29, 'a', 22, 3.14, 35, 'b', 'ISEN Bretagne', 56, 44]
[29, 'a', 22, 3.14, 35, 'b', 56, 44]
```

### 1.14 La méthode *split()*

La méthode ***split*** est très pratique, elle permet (par exemple) de créer une liste de mots à partir d’une chaîne de caractères. La transformation en liste permettra ensuite de remplacer / ajouter / supprimer facilement des éléments. Voici un exemple :

```
[27]: phrase = "toto est beau"
# Par défaut le séparateur est l'espace,
# mais on peut préciser un autre caractère
# à la méthode split
mots_liste = phrase.split()
print(mots_liste)
print(type(mots_liste))

chaine = "fait,beau,fait,pas,beau"
mots_liste = chaine.split(",")
print(mots_liste)
```

```
['toto', 'est', 'beau']
<class 'list'>
['fait', 'beau', 'fait', 'pas', 'beau']
```

### 1.15 La méthode *join()*

C’est la “réciproque” de la méthode *split*. Elle permet, à partir d’une liste de chaînes de reconstituer une chaîne de caractères.

Exemple :

```
[2]: mots_liste = ['toto', 'est', 'beau']
phrase = " ".join(mots_liste) # Noter l'espace entre les guillemets
print(phrase)
```

toto est beau

## 1.16 Mini-TP : vérifier la validité d'une date

- Objectif : demander à l'utilisateur de saisir un jour, un mois et une année. Lui indiquer si la date est valide ou non. Tant que la date est invalide recommencer.
- Indications :
  - Il sera utile de savoir si une année est bissextile ou non : dans ce cas il y a un 29 février. Une année est bissextile si l'année est divisible par 4 et que cette année est divisible par 400 mais pas par 100.
  - Avant d'écrire le programme en Python, écrire la structure du programme avec des pseudo-instructions.

```
[14]: while True:
    jour = int(input("Jour : "))
    mois = int(input("Mois : "))
    annee = int(input("Année : "))
    if jour > 31 or jour < 1 or annee < 0 or mois < 1 or mois > 12:
        print("Date invalide")
    elif mois in [1, 3, 5, 7, 8, 10, 12]:
        break
    else: # Mois à 30 jours ou février
        if mois == 2:
            if annee%4 == 0 and (annee%400 == 0 or not annee%100 == 0): #
↪Année bissextile
                if jour > 29:
                    print("Le mois de février d'une année bissextile ne peut
↪avoir plus de 29 jours")
                else:
                    break
            else: # Année non bissextile
                if jour > 28:
                    print("Le mois de février d'une année non bissextile ne
↪peut avoir plus de 28 jours")
                else:
                    break
        else:
            if jour > 30:
                print("Erreur, pour ce mois il ne peut y avoir plus de 30
↪jours")
            else:
                break
    print("Date valide !")
```

Jour : 12  
Mois : 4  
Année : -1

```

Date invalide
Jour : 31
Mois : 4
Année : 2018
Erreur, pour ce mois il ne peut y avoir plus de 30 jours
Jour : 29
Mois : 2
Année : 2017
Le mois de février d'une année non bissextile ne peut avoir plus de 28 jours
Jour : 29
Mois : 2
Année : 2016
Date valide !

```

## 1.17 Mini-TP : manipulation de matrices

Soit la matrice :

```

1 4 2
5 4 4
8 2 1

```

- 1) Ajouter +1 pour chaque élément
- 2) Mettre 1 sur les éléments de la diagonale

- Indications :

- Pour représenter une matrice en Python, on peut utiliser des listes de listes : `matrice = [[1, 4, 2], [5, 4, 4], [8, 2, 1]]`
- Pour vérifier que la matrice a bien le résultat attendu, on peut utiliser l'instruction `assert` : `assert matrice == [[2, 5, 3], [6, 5, 5], [9, 3, 2]]` (pour la question 1)

```

[16]: # Question 1
matrice = [[1, 4, 2], [5, 4, 4], [8, 2, 1]]
i = 0
j = 0
for ligne in matrice:
    for elem in ligne:
        matrice[i][j] += 1
        j += 1
    i += 1
    j = 0

assert matrice == [[2, 5, 3], [6, 5, 5], [9, 3, 2]]

# Question 2
i = 0
j = 0
for ligne in matrice:

```

```

for elem in ligne:
    if i == j:
        matrice[i][j] = 1
    j += 1
i += 1
j = 0

assert matrice == [[1, 5, 3], [6, 1, 5], [9, 3, 1]]

```

## 1.18 Mini-TP : le tri à bulles

Le principe du tri bulle est de comparer deux à deux les éléments consécutifs d'une liste ( $e1$  et  $e2$ ) et d'effectuer une permutation si  $e1 > e2$ . On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

Soit la liste = [4, 2, 8, 1, 7, 5, 3, 6]

```

[2, 4, 8, 1, 7, 5, 3, 6]
[2, 4, 1, 8, 7, 5, 3, 6]
[2, 4, 1, 7, 8, 5, 3, 6]
[2, 4, 1, 7, 5, 8, 3, 6]
[2, 4, 1, 7, 5, 3, 8, 6]
[2, 4, 1, 7, 5, 3, 6, 8]
Fin traitement liste
[2, 1, 4, 7, 5, 3, 6, 8]
[2, 1, 4, 5, 7, 3, 6, 8]
[2, 1, 4, 5, 3, 7, 6, 8]
[2, 1, 4, 5, 3, 6, 7, 8]
Fin traitement liste
[1, 2, 4, 5, 3, 6, 7, 8]
[1, 2, 4, 3, 5, 6, 7, 8]
Fin traitement liste
[1, 2, 3, 4, 5, 6, 7, 8]
Fin traitement liste

```

On voit qu'à l'étape 13, la liste est triée, c'est donc terminé !

- 1) Réfléchir à un algorithme permettant de réaliser ces étapes, à savoir la permutation des deux éléments  $e1$  et  $e2$ , lorsque  $e1$  est supérieur à  $e2$ . On veillera à ce qu'il soit générique, c'est à dire qu'il puisse s'appliquer à n'importe quelle liste.
- 2) Ecrire en Python cet algorithme.
- 3) Pourquoi cet algorithme (un classique de l'apprentissage de la programmation) s'appelle le tri à bulles ?

```

[1]: liste = [4, 2, 8, 1, 7, 5, 3, 6]
permutation = True
i = 0
while permutation:
    permutation = False

```

```

    for elem in liste:
        if i < len(liste)-1:
            if liste[i] > liste[i+1]:
                liste[i+1], liste[i] = liste[i], liste[i+1] # Echange de
↪valeurs
                permutation = True
                print(liste) # Juste pour contrôler ce qui se passe
            i += 1
    print("Fin traitement liste")
    i = 0

```

```

[2, 4, 8, 1, 7, 5, 3, 6]
[2, 4, 1, 8, 7, 5, 3, 6]
[2, 4, 1, 7, 8, 5, 3, 6]
[2, 4, 1, 7, 5, 8, 3, 6]
[2, 4, 1, 7, 5, 3, 8, 6]
[2, 4, 1, 7, 5, 3, 6, 8]
Fin traitement liste
[2, 1, 4, 7, 5, 3, 6, 8]
[2, 1, 4, 5, 7, 3, 6, 8]
[2, 1, 4, 5, 3, 7, 6, 8]
[2, 1, 4, 5, 3, 6, 7, 8]
Fin traitement liste
[1, 2, 4, 5, 3, 6, 7, 8]
[1, 2, 4, 3, 5, 6, 7, 8]
Fin traitement liste
[1, 2, 3, 4, 5, 6, 7, 8]
Fin traitement liste
Fin traitement liste

```

```

[1]: liste = [4, 2, 8, 1, 7, 5, 3, 6]
    permutation = True
    i = 0
    while permutation:
        permutation = False
        for elem in liste:
            if i < len(liste)-1:
                if liste[i] > liste[i+1]:
                    liste[i+1], liste[i] = liste[i], liste[i+1] # Echange de
↪valeurs
                    permutation = True
                    print(liste) # Juste pour contrôler ce qui se passe
                i += 1
        print("Fin traitement liste")
        i = 0

```

```

[2, 4, 8, 1, 7, 5, 3, 6]

```

```

[2, 4, 1, 8, 7, 5, 3, 6]
[2, 4, 1, 7, 8, 5, 3, 6]
[2, 4, 1, 7, 5, 8, 3, 6]
[2, 4, 1, 7, 5, 3, 8, 6]
[2, 4, 1, 7, 5, 3, 6, 8]
Fin traitement liste
[2, 1, 4, 7, 5, 3, 6, 8]
[2, 1, 4, 5, 7, 3, 6, 8]
[2, 1, 4, 5, 3, 7, 6, 8]
[2, 1, 4, 5, 3, 6, 7, 8]
Fin traitement liste
[1, 2, 4, 5, 3, 6, 7, 8]
[1, 2, 4, 3, 5, 6, 7, 8]
Fin traitement liste
[1, 2, 3, 4, 5, 6, 7, 8]
Fin traitement liste
Fin traitement liste

```

[3]: *# Une autre version...*

```

liste = [4, 2, 8, 1, 7, 5, 3, 6]
lg_liste = len(liste)
# On parcourt toute la liste
for i in range(lg_liste):
    for j in range(0, lg_liste-i-1):
        # échanger si l'élément trouvé est plus grand que le suivant
        if liste[j] > liste[j+1] :
            liste[j], liste[j+1] = liste[j+1], liste[j]
            print(liste)
print("fin boucle 2")

```

```

[2, 4, 8, 1, 7, 5, 3, 6]
[2, 4, 1, 8, 7, 5, 3, 6]
[2, 4, 1, 7, 8, 5, 3, 6]
[2, 4, 1, 7, 5, 8, 3, 6]
[2, 4, 1, 7, 5, 3, 8, 6]
[2, 4, 1, 7, 5, 3, 6, 8]
fin boucle 2
[2, 1, 4, 7, 5, 3, 6, 8]
[2, 1, 4, 5, 7, 3, 6, 8]
[2, 1, 4, 5, 3, 7, 6, 8]
[2, 1, 4, 5, 3, 6, 7, 8]
fin boucle 2
[1, 2, 4, 5, 3, 6, 7, 8]
[1, 2, 4, 3, 5, 6, 7, 8]
fin boucle 2
[1, 2, 3, 4, 5, 6, 7, 8]
fin boucle 2

```

```

fin boucle 2
fin boucle 2
fin boucle 2
fin boucle 2

```

## 1.19 Mini-TP le jeu du pendu

### 1.19.1 Indications générales

- On définit une liste de mots à trouver.
- On fixe un nombre de tentatives maximum.
- On choisit un mot au hasard dans la liste de mots (voir la fonction `random.choice`).
- On demande à l'utilisateur de saisir une lettre.
- Si la lettre a déjà été saisie, on lui indique.
- Le nombre de tentatives est incrémenté si la lettre n'appartient pas au mot (et si la lettre n'a pas été déjà proposée).
- A chaque tour on affiche sa réponse en cours (lettres devinées), le nombre de tentatives et les lettres proposées.
- A la fin on affiche à l'utilisateur s'il a gagné ou s'il a perdu (quand il a dépassé le nombre de tentatives).

### 1.19.2 Indications avancées

- Algorithme général.
- Stockage de la réponse en cours : dans une liste.
- Utiliser la fonction `join` pour comparer la réponse courante et le mot à trouver.

```

[14]: import random

MOTS = ("python", "java", "isen", "brest")
NB_TENTATIVES_MAX = 5

# Initialisations
nb_tentatives = 0
gagne = False
lettres_proposees = []
mot_a_trouver = random.choice(MOTS)
print(mot_a_trouver)
# Initialisation de la réponse
reponse_crte = ['-'] * len(mot_a_trouver)

while not gagne and nb_tentatives < NB_TENTATIVES_MAX:
    print(f"{reponse_crte} / Tentatives = {nb_tentatives} / Lettres proposées : {lettres_proposees}")
    lettre = input("Lettre ? ")
    lettre = lettre.lower()
    if len(lettre) > 1:
        print("Merci d'entrer une lettre")
    elif lettre in lettres_proposees:

```



```

        print("Vous avez déjà proposé cette lettre")
    else:
        lettres_proposees.append(lettre)
        if lettre in mot_a_trouver:
            for i, car in enumerate(mot_a_trouver):
                if car == lettre:
                    reponse_crte[i] = lettre
            gagne = "".join(reponse_crte) == mot_a_trouver
        else:
            nb_tentatives += 1

if gagne:
    print("Bravo, vous avez gagné !")
else:
    print ("Vous avez perdu, la réponse était : ", mot_a_trouver)

```

```

python
['-', '-', '-', '-', '-', '-'] / Tentatives = 0 / Lettres proposées : []
Lettre ? p
['p', '-', '-', '-', '-', '-'] / Tentatives = 0 / Lettres proposées : ['p']
Lettre ? k
['p', '-', '-', '-', '-', '-'] / Tentatives = 1 / Lettres proposées : ['p', 'k']
Lettre ? t
['p', '-', 't', '-', '-', '-'] / Tentatives = 1 / Lettres proposées : ['p', 'k', 't']
Lettre ? o
['p', '-', 't', '-', 'o', '-'] / Tentatives = 1 / Lettres proposées : ['p', 'k', 't', 'o']
Lettre ? h
['p', '-', 't', 'h', 'o', '-'] / Tentatives = 1 / Lettres proposées : ['p', 'k', 't', 'o', 'h']
Lettre ? b
['p', '-', 't', 'h', 'o', '-'] / Tentatives = 2 / Lettres proposées : ['p', 'k', 't', 'o', 'h', 'b']
Lettre ? q
['p', '-', 't', 'h', 'o', '-'] / Tentatives = 3 / Lettres proposées : ['p', 'k', 't', 'o', 'h', 'b', 'q']
Lettre ? z
['p', '-', 't', 'h', 'o', '-'] / Tentatives = 4 / Lettres proposées : ['p', 'k', 't', 'o', 'h', 'b', 'q', 'z']
Lettre ? y
['p', 'y', 't', 'h', 'o', '-'] / Tentatives = 4 / Lettres proposées : ['p', 'k', 't', 'o', 'h', 'b', 'q', 'z', 'y']
Lettre ? y
Vous avez déjà proposé cette lettre
['p', 'y', 't', 'h', 'o', '-'] / Tentatives = 4 / Lettres proposées : ['p', 'k', 't', 'o', 'h', 'b', 'q', 'z', 'y']

```

Lettre ? v

Vous avez perdu, la réponse était : python

## 1.20 Mini-TP matplotlib

La bibliothèque matplotlib permet de tracer des graphique (et bien plus !). Voici un exemple :

```
import matplotlib.pyplot as plt
```

```
fig = plt.figure ()
```

```
# Initialisation des listes
```

```
liste_x = [0 , 1.5 , 2 , 3]
```

```
liste_y = [ -5 , 18 , 5.4 , 6]
```

```
# Affichage du graphique
```

```
plt.plot (liste_x, liste_y)
```

```
[19]: %matplotlib inline
import matplotlib . pyplot as plt

fig = plt.figure ()

# Initialisation des listes
liste_x = [0 , 1.5 , 2 , 3]
liste_y = [ -5 , 18 , 5.4 , 6]

# Affichage du graphique
plt.plot (liste_x, liste_y)
```

```
[19]: [<matplotlib.lines.Line2D at 0x7f2a67c74a30>]
```

### 1.20.1 Exercice : graphique des températures

- Pour chaque mois de l'année, demander à l'utilisateur de saisir une température.
- Afficher ensuite le graphique correspondant.

```
[ ]: %matplotlib inline
import matplotlib . pyplot as plt

fig = plt.figure ()

liste_mois = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']
# ou : liste_mois = [str(mois) for mois in range(1, 13)]

liste_temperatures = []

for mois in liste_mois:
    #print("Mois : ", mois, end=' ')
    temperature = float(input(f"Mois {mois} / Température : "))
    liste_temperatures.append(temperature)

# Graphique
plt.plot (liste_mois, liste_temperatures)
```



# 05-cours\_python\_chaines\_car

January 13, 2020

## 1 Manipulations de base sur les chaînes de caractères

### 1.1 Définition

Une chaîne peut se définir approximativement comme une suite de caractères. On utilise comme délimiteurs les caractères " ou '.

```
msg1 = "Bonjour"
msg2 = 'Bonjour'
msg3 = "J'aime"
msg4 = 'J'aime' # Erreur
```

### 1.2 Banaliser les caractères

**Exercice :** “banaliser” un caractère dans une chaîne

Afficher les phrases suivantes :

- Python c’est facile
- “facile” selon le prof !
- Python c’est “facile”

```
[1]: msg = "Python c'est facile"
      print(msg)
      msg = '"facile" selon le prof !'
      print(msg)
      msg = "Python c'est \"facile\""
      print(msg)
```

```
Python c'est facile
"facile" selon le prof !
Python c'est "facile"
```

### 1.3 Chaînes sur plusieurs lignes

```
print("""Bonjour,
J'espère que vous allez bien !
""")
```

**Exercice :** utilisation de variables

En utilisant deux variables (une pour le caractère '#', une pour le message) afficher à l’écran :

```
#####
Bienvenue
Degemer Mat
Willkommen
#####
```

```
[1]: cadre = "#####"
message = ""
Bienvenue
Degemer Mat
Willkommen
""
print(f"{cadre}{message}{cadre}") # ou bien : print(cadre + message + cadre)
```

```
#####
Bienvenue
Degemer Mat
Willkommen
#####
```

## 1.4 Autres “bizarreries”

Que fait le code suivant ?

```
msg = "#" * 80
print(msg)
```

```
[5]: msg = "#" * 80
print(msg)
```

```
#####
```

## 1.5 Indicage, extraction, longueur

Assignons la chaîne de caractères ISEN Bretagne à une variable :

```
isen_bretagne = "ISEN Bretagne"
```


### 1.5.1 Indicage

Une chaîne de caractères étant une séquence (ou suite) de caractères, il est possible d’accéder à un caractère particulier, connaissant son indice (sa position dans la chaîne). Sachant que l’indexation commençant à 0, on a donc :

```
[3]: isen_bretagne = "ISEN Bretagne"
print(isen_bretagne[0])
print(isen_bretagne[4]) # Oui, l'espace est un caractère
print(isen_bretagne[5])
print(isen_bretagne[20])
```

I

B

```
↳ -----  
↳ IndexError                                Traceback (most recent call↳  
↳ last)  
  
    <ipython-input-3-f8e34f9c1c5b> in <module>  
      3 print(isen_bretagne[4]) # Oui l'espace est un caractère  
      4 print(isen_bretagne[5])  
----> 5 print(isen_bretagne[20])  
  
IndexError: string index out of range
```

## 1.6 Extraction d'une sous-chaîne

Pour extraire des morceaux de la chaîne de caractères, on procède de la façon suivante :

```
sous_chaine1 = chaine[m:n]
```

**Attention !** : le morceau de chaîne contiendra les caractères de la chaîne originale depuis l'index *m* **inclus** jusqu'à l'index *n* **exclu** (en d'autres termes, de l'index *m* à l'index *n-1* tous deux inclus).

```
[6]: chaine = "ISEN Bretagne"  
     print(chaine[0:4])  
     print(chaine[6:8])
```

```
ISEN  
re
```

A noter qu'on peut omettre les index de début ou de fin. Effectuer quelques tests du type :

```
[4]: chaine = "ISEN Bretagne"  
     sous_chaine1 = chaine[:4]  
     sous_chaine2 = chaine[5:]  
     print(f"{sous_chaine1} {sous_chaine2}")
```

```
ISEN Bretagne
```

### 1.6.1 Longueur d'une chaîne

Il est possible de connaître la longueur de la chaîne de caractères en utilisant la fonction *len()* :

```
longueur_chaine = len(string_isen)
print(longueur_chaine)
```

### 1.7 Exercice : extraire des morceaux d’une chaîne de caractères

Extraire les suites de caractères *ISEN* et *retag* à partir de la chaîne de caractères “ISEN Bretagne”. Placer le morceaux de chaînes dans deux variables *chaine1* et *chaine2* puis afficher ces deux variables.

```
[5]: chaine_isen = "ISEN Bretagne"
chaine1 = chaine[0:4]
chaine2 = chaine[6:11]
print(chaine1)
print(chaine2)
```

ISEN retag

### 1.8 Exercice : concaténation de sous-chaînes

Extraire les deux mots *ISEN* et *Bretagne* à partir de la chaîne d’origine “ISEN Bretagne”. En utilisant la concaténation de chaînes de caractères, reconstituer ensuite la chaîne ISEN Bretagne.

```
[6]: # 1ère possibilité
chaine = "ISEN Bretagne"
mot1 = chaine[:4]
mot2 = chaine[5:]
# 1ère possibilité
chaine2 = mot1 + " " + mot2
print(chaine2)
# 2ème possibilité (mieux)
chaine2 = f"{mot1} {mot2}"
print(chaine2)
```

ISEN Bretagne  
ISEN Bretagne

### 1.9 Exercice : extraire les voyelles d’une chaîne

- Demander un mot à l’écran (en minuscules)
- Parcourir ce mot
- A la fin, afficher la liste des voyelles

```
[15]: # 1ère version
mot = "python"
i = 0
voyelles = ""
while i < len(mot):
    if mot[i] == 'a' or mot[i] == 'e' or mot[i] == 'i' or mot[i] == 'o' or
    ↪ mot[i] == 'u' or mot[i] == 'y':
```



```

        voyelles += mot[i]
    i+=1
print(voyelles)

```

yo

```

[16]: # 2ème version : plus élégante
mot = "python"
i = 0
voyelles = ""
while i < len(mot):
    if mot[i] in "aeiouy":
        voyelles += mot[i]
    i+=1
print(voyelles)

```

yo

## 1.10 Quelques opérations possibles sur les chaînes de caractères

Il existe de nombreuses possibilités pour effectuer des opérations sur des chaînes de caractères :

- Mise en minuscules / majuscules
- Suppression des espaces au début et à la fin
- Recherche, remplacement...

Les chaînes de caractères disposent de méthodes (~fonction) qui vont permettre ces opérations.

Voici quelques exemples :

```

[1]: #
# Mise en majuscules
#
my_str = "sacré toto"
# upper() est une méthode (~fonction) de l'objet str
my_str_upper = my_str.upper()
print(my_str) # my_str n'est pas modifiée
print(my_str_upper)
# On n'est (évidemment) pas obligé de passer par une autre variable
my_str = my_str_upper

#
# Suppression des espaces
#
my_str2 = "    fait pas beau    "
print(f"***{my_str2}***")
my_str2 = my_str2.strip()
print(f"***{my_str2}***")

```

```
sacré toto
SACRÉ TOTO
***    fait pas beau    ***
***fait pas beau***
```

### 1.10.1 Exercice : trouver deux façons de tester si une chaîne de caractères est vide

```
[ ]: # 1ère façon
str_value1 = input("Saisissez une chaîne de caractères ")
if str_value1.strip() == '':
    print("Chaîne vide")
else:
    print("Chaîne NON vide")

# 2ème façon
str_value2 = input("Saisissez une chaîne de caractères ")
if len(str_value2.strip()) == 0:
    print("Chaîne vide")
else:
    print("Chaîne NON vide")
```

### 1.10.2 Exercice : demander à l'utilisateur de saisir une réponse :

- S'il répond "oui" afficher "On continue !"
- Sinon afficher "Stop !"

On ne doit pas tenir compte des majuscules / minuscules dans la réponse (on dit qu'on "ignore la casse").

```
[8]: reponse = input("Réponse (oui / non) : ")
reponse = reponse.strip() # Suppression des espaces éventuels avant et après
reponse = reponse.lower() # On met tout en minuscules
# On aurait pu faire aussi une seule opération : reponse = reponse.strip().
# → lower()

if reponse == "oui":
    print("On continue !")
else:
    print("Stop !")
```

```
Réponse (oui / non) : Oui
On continue !
```

```
[ ]: ## Exercice : chaîne de caractères et liste
Quelle est la différence entre ces deux codes ?
python
liste_car = ['b', 'o', 'n', 'j', 'o', 'u', 'r', ' ', 'I', 'S', 'E', 'N']
for c in liste_car:
```

```

    print(c)
~ ~ ~

et :

~ ~ ~python
message = "bonjour ISEN"
for c in message:
    print(c)
~ ~ ~

Que se passe t-il si on effectue :

~ ~ ~python
liste_car[7] = '-'
message[7] = '-'
~ ~ ~

```

### 1.11 Exercice : former une chaîne de caractères à partir d'une liste

- Reformuler une chaîne de caractères à partir de la liste ci-dessous en utilisant la fonction *join* :

```

# ["toto", "est", "beau", "quand", "il", "le", "faut"]
# Résultat attendu : "toto est beau quand il le faut"

```

```

[ ]: my_list = ["toto", "est", "beau", "quand", "il", "le", "faut"]
my_str = " ".join(my_list)
print(my_str)

```

## 2 Exercices de révision

### 2.1 Concaténation de chaînes de caractères

Afin de mettre en place un système de caisses automatiques, le directeur du cinéma *Les Studios* vous demande de réaliser un programme demandant au client son âge, s'il est abonné et lui annonçant en retour le tarif appliqué :

- Moins de 18 ans : 4 €
- Abonné : 5,50 €
- Sinon : 8 €

Afficher selon les valeurs saisies par le client, par exemple :

- Vous avez 12 ans, le montant à régler est de 4 €
- Vous êtes abonné, le montant à régler est de 5,50 €

```
[7]: age = int(input("Quel est votre âge ? "))
    abonne = 'n'
    if age < 18:
        prix = 4.00
    else:
        abonne = input("Etes-vous abonné ? (o/n) ")
        if abonne.lower() == 'o':
            prix = 5.50
        else:
            prix = 8.00

    if abonne.lower() == 'o':
        print(f"Vous êtes abonné, le montant à régler est de {prix} €")
    else:
        print(f"Vous avez {age} ans, le montant à régler est de {prix} €")
```

Quel est votre âge ? 12

Vous avez 12 ans, le montant à régler est de 4.0 €

## 2.2 Chaînes de caractères et conditions

Le cinéma “Liberté” étant proche de la faillite, à cause notamment de la forte concurrence des Studios, les dirigeants décident de mener une politique agressive sur les tarifs. Après avoir réfléchi toute la nuit ils vous communiquent la façon dont la nouvelle version du logiciel doit calculer les prix :

- Demander à l'utilisateur son âge
- Si il a entre 0 et 4 ans ou plus de 80 ans, c'est gratuit
- Si il a plus de 100 ans on lui donne 5 €
- Si il a entre 5 et 17 ans il paye 3 € et en plus on lui offre un seau de Popcorn
- Si il a 18 ans ou plus et qu'il habite Brest il paye 7 €. S'il n'habite pas Brest c'est 8 €
- Ah au fait, quelque soit son âge, s'il habite Quimper c'est 50 €
- Le programme doit fonctionner de la même façon s'il y a des majuscules / minuscules dans le nom de la ville.

```
[1]: # Solution 1

ville = input("Où habitez-vous ? ")
ville = ville.strip().lower()

if ville == 'quimper':
    print("Euh... c'est 50 €")
else:
    age = int(input("Votre âge ? "))
    if age < 5:
```

```

    print("C'est gratuit !")
elif age > 100:
    print(f"On vous rembourse : 5 Euro")
elif age >= 5 and age <= 17:
    print("Tiens un seau de popcorn ! Mais tu dois régler 3 €")
else: # L'utilisateur a 18 ans ou plus (mais <= 100 ans)
    if ville == 'brest':
        print("Vous devez payer 7 €")
    else:
        print("Vous devez payer 8 €")

```

Où habitez-vous ? brest

Votre âge ? 12

Tiens un seau de popcorn ! Mais tu dois régler 3 Euro

[2]: # Solution 2

```

ville = input("Où habitez-vous ? ")
ville = ville.strip().lower()

if ville == 'quimper':
    tarif = 50.0
else:
    tarif = 0.0
    age = int(input("Votre âge ? "))
    if not ((age > 0 and age <=4) or (age > 80)):
        # L'utilisateur n'a pas entre 0 et 4 ans et n'a pas plus de 80 ans
        if age > 100:
            tarif = -5.0
        elif age >= 5 and age <= 17:
            print("Tiens un seau de popcorn !")
            tarif = 3.0
        else:
            # L'utilisateur a plus de 17 ans (mais n'a pas plus de 80 ans)
            if ville == 'brest':
                tarif = 7.0
            else:
                tarif = 8.0
        # Si la condition not ((age > 0 and age <=4) or (age > 80)) n'est pas
        ↪ respectée
        # alors tarif = 0.0 (valeur mise au départ)
    if tarif == 0.0:
        print("C'est gratuit !")
    elif tarif > 0:
        print(f"Vous devez régler : {tarif} Euro")
    else:
        print(f"On vous rembourse : {tarif} Euro")

```

Où habitez-vous ? brest  
Votre âge ? 12  
Tiens un seau de popcorn !  
Vous devez régler : 3.0 Euro

# 06-cours\_\_python\_\_fonctions

February 21, 2020

## 1 Les fonctions

### 1.1 Présentation

#### 1.1.1 Qu'est-ce qu'une fonction ?

Une fonction est un bloc d'un programme qui contient une série d'instructions.

- Une fonction peut être appelée n'importe où dans un programme.
- Les instructions contenues dans la fonction ne sont exécutées qu'à l'appel de la fonction.

Nous avons déjà utilisé des fonctions :

- `len(liste)` -> renvoie la longueur d'une liste
- `input(chaine)` -> renvoie une valeur saisie par l'utilisateur
- `print(valeur)` -> affiche une valeur à l'écran (mais ne renvoie rien)

Ces fonctions sont disponibles par défaut dans le langage Python.

#### 1.1.2 Ecriture d'une fonction

Outre l'utilisation des fonctions fournies par le langage, il est également possible pour le programmeur de définir des fonctions. Les avantages sont les suivants :

- Le code d'un programme peut être écrit par petits blocs, ce qui permet d'isoler correctement les différentes parties et de les modifier simplement. Cela est particulièrement utile lorsque le programme comporte de nombreuses lignes.
- Lorsqu'on répète une série d'instructions, on peut les isoler dans une fonction et l'appeler lorsqu'on en a besoin ailleurs dans le programme.

Une fonction s'écrit de la façon suivante :

```
def ma_fonction1():  # Fonction sans paramètre
    # Série d'instructions (indentées)
    ...

def ma_fonction2(a, b):  # Fonction avec deux paramètres
    ...
```

**Exemple : écriture d'une fonction mathématique** On peut aussi faire une analogie avec les fonctions mathématiques : on donne une ou des valeur(s) en entrée et on obtient une valeur en sortie. Par exemple, une fonction qui calcule le double d'un nombre donné s'écrit :  $f(x) = 2x$ . Si

on veut savoir quel est le double de 3, nous appellerons la fonction  $f$  en lui indiquant que  $x$  vaut 3 :  $f(3)$ . Le résultat, la valeur de retour de la fonction sera  $2 \times 3$  soit 6. Commençons donc par écrire cette fonction en Python :

```
def f(x):  
    resultat = 2*x  
    return resultat
```

ou

```
def f(x):  
    return 2*x
```

La partie de définition du nom de la fonction et des paramètres (valeurs) qui lui seront transmis, se fait sur la première ligne par `def f(x):`. Notez que, comme pour les tests de conditions ou pour les boucles, nous débutons un nouveau bloc : toutes les instructions qui seront indentées (décalées par rapport à la marge de gauche), feront partie de la fonction.

La valeur de retour de la fonction est définie par l'instruction `return` (ici, ce sera  $2*x$ ). Quand nous appellerons cette fonction, la valeur de retour sera calculée, renvoyée à la ligne de code ayant effectué l'appel, puis utilisée.

A noter que les instructions contenues dans la fonction **ne seront exécutées que lorsqu'on fera appel à la fonction**.

```
def f(x):  
    # Contenu de la fonction  
    return 2*x  
  
# A partir d'ici le code se trouve hors de la fonction  
# On appelle la fonction (et son code est exécuté)  
f(3)  
# Mais rien n'est visible à l'écran, car on ne demande pas d'affichage...
```

Si on souhaite un affichage:

```
def f(x):  
    # Contenu de la fonction  
    return 2*x
```

```
y = f(3)  
print("Résultat : ", y)
```

Notons que si on ne souhaite pas stocker dans une variable le résultat de la fonction, on faire l'appel de la fonction directement dans la fonction `print`.

```
def f(x):  
    # Contenu de la fonction  
    return 2*x
```

```
print("Résultat : ", f(3))
```

Remarque : la transmission d'un paramètre à une fonction peut se faire également par une variable, par exemple :



```
def f(x):
    # Contenu de la fonction
    return 2*x

a = 3
print("Résultat : ", f(a))
```

**Exercice : retour et affichage de valeur** Sans exécuter le programme ci-dessous, indiquez ce qu'affichent les instructions *print* ci-dessous :

```
def fonction_polie():
    return "Bonjour"

# Programme principal
print("Début")
valeur = fonction_polie()
print("Appel #1", valeur)
print("Appel #2", fonction_polie())
fonction_polie()
print("C'est fini")
```

## 1.2 Remarque (très importante) sur l'organisation d'un programme

Par convention l'ordre d'un programme est le suivant :

```
# Ajout des import éventuels
import ...

def ma_fonction1():
    ...

def ma_fonctionN():
    ...

# Programme "principal"
print("Bonjour je suis en forme")
ma_fonction1()
ma_fonctionN()
print("Ouf c'est fini !")
```

Si on écrit simplement :

```
def ma_fonction1():
    ...

def ma_fonctionN():
    ...
```

Il ne se passe rien, car lorsque l'interpréteur Python exécute le code, **il n'exécute pas les fonctions tant qu'elles ne sont pas appelées.**

Que se passe t-il à l'exécution du programme suivant ?

```
def bonjour():
    print("Bonjour")

print("Mais que diable fais-je ici ?")

def aurevoir():
    print("Hasta luego")

Et pour ce programme ?

def bonjour():
    print("Bonjour")

print("Mais que diable fais-je ici ?")

def aurevoir():
    print("Hasta luego")

bonjour()
aurevoir()
```

### 1.3 Fonctions sans valeur de retour (procédure)

En informatique, une fonction ne prend pas forcément de paramètre(s) et ne renvoie pas forcément une valeur... Voici un exemple :

```
def ma_fonction():
    print("Bonjour")

ma_fonction() # Appel de la fonction et affichage de bonjour
```

Remarque : une fonction qui ne comporte pas d'instruction return renvoie None par défaut, ce code est donc équivalent :

```
def ma_fonction():
    print("Bonjour")
    return None

ma_fonction()
```

### 1.4 Commentaire d'une fonction

Il est souvent judicieux de mettre une description de la fonction dans son entête. Pour cela, on va utiliser la possibilité en Python d'afficher une chaîne de caractères sur plusieurs lignes. C'est ce qu'on appelle la "docstring" de la fonction.

```
def racine_carree(valeur):
    """Fonction de calcul de la racine carrée
    @param valeur: la valeur dont on souhaite extraire la racine
    @return: la racine de la valeur ou None si le nombre est négatif
```

```

"""
# Code de la fonction
# ...

```

## 1.5 Exercice : écriture d'une fonction de calcul de racine carrée

Ecrire la fonction `racine_carree(valeur)`. Elle prend en paramètre une valeur et en retourne la racine carrée si la valeur est positive et `None` dans le cas contraire. On prendra soin d'écrire la *docstring* de la fonction.

```

[ ]: def racine(valeur):
    """Fonction de calcul de la racine carrée
    valeur: La valeur dont on souhaite extraire la racine
    return: La racine de la valeur ou None si le nombre est négatif
    """
    if valeur >= 0:
        return valeur**0.5
    else:
        return None

# Programme principal
print(racine(4))
print(racine(-1))

```

```

[ ]: # Remarque : on peut vérifier automatiquement le retour des valeurs avec
    ↪ l'instruction assert
# Cela est particulièrement utile lorsqu'il y a beaucoup de cas à vérifier
def racine(valeur):
    """Fonction de calcul de la racine carrée
    valeur: La valeur dont on souhaite extraire la racine
    return: La racine de la valeur ou None si le nombre est négatif
    """
    if valeur >= 0:
        return valeur**0.5
    else:
        return None

# Vérification
assert racine(4) == 2
assert racine(-1) == None
## Une erreur Python va être déclenchée
assert racine(6) == 2

```

## 1.6 Exercices : appel de différentes fonctions de calcul

- Exercice 1 : écrire une fonction `affiche_surface_cercle` qui prend en paramètre le rayon et affiche la surface du cercle

- Exercice 2 : écrire une fonction `calcul_surface_cercle` qui prend en paramètre le rayon et renvoie la surface du cercle
- Exercice 3 : écrire une fonction `calcul_volume_cylindre` qui prend en paramètre le rayon du cercle ainsi que la hauteur et qui renvoie le volume

```
[ ]: # Exercice 1

import math

def affiche_surface_cercle(rayon_cm):
    val_arr = round(rayon_cm**2 * math.pi, 2)
    print(val_arr, "cm2")

# Programme principal
affiche_surface_cercle(20)
```

```
[ ]: # Exercice 2

import math

def calcul_surface_cercle(rayon_cm):
    return rayon_cm**2 * math.pi

# Programme principal
surface_cm2 = calcul_surface_cercle(20)
print(surface_cm2, "cm2")
```

```
[ ]: # Exercice 3 - v1

import math

def calcul_volume_cylindre(rayon_cm, h_cm):
    return rayon_cm**2 * math.pi * h_cm

# Programme principal
volume_cm3 = calcul_volume_cylindre(20, 10)
print(volume_cm3, "cm3")
```

```
[ ]: # Exercice 3 - v2

import math

def calcul_surface_cercle(rayon_cm):
    return rayon_cm**2 * math.pi

def calcul_volume_cylindre(rayon_cm, h_cm):
    # Question : que se passerait-il si on appelait
```

```

    # affiche_surface_cercle au lieu de calcul_surface_cercle
    # dans la ligne ci-dessous ?
    return calcul_surface_cercle(rayon_cm) * h_cm

# Programme principal
volume_cm3 = calcul_volume_cylindre(20, 10)
print(volume_cm3, "cm3")

```

## 1.7 Exercice : calcul de l'écart-type d'une liste de nombres

- Calculer l'écart-type pour les listes suivantes (ne pas utiliser le module *statistics*) :

```

liste1 = [10, 11, 2, 5, 13]
liste2 = [1, 18, 20, 10]
liste3 = [10, 10, 10, 10]

```

- Ecrire une fonction `ecart_type` qui prend en entrée une liste de valeurs et qui retourne l'écart-type
- Dans le programme principal, effectuer les appels avec `liste1`, `liste2` et `liste3` et afficher le résultat

```

[ ]: # V1
def ecart_type(valeurs):
    somme = 0
    for elem in valeurs:
        somme += elem
    moyenne = somme / len(valeurs)
    somme_carre = 0
    for v in valeurs:
        somme_carre += (v - moyenne)**2
    return (somme_carre/len(valeurs))*0.5

# Programme principal
liste1 = [10, 11, 2, 5, 13]
liste2 = [1, 18, 20, 10]
liste3 = [10, 10, 10, 10]
ecart_type_1 = ecart_type(liste1)
ecart_type_2 = ecart_type(liste2)
ecart_type_3 = ecart_type(liste3)
print(ecart_type_1, ecart_type_2, ecart_type_3)
# Ou sans variable intermédiaire
print(ecart_type(liste1), ecart_type(liste2), ecart_type(liste3))

```

```

[ ]: # V2
import math

def ecart_type(valeurs):
    moyenne = sum(valeurs) / len(valeurs)

```

```

total = 0
for v in valeurs:
    total += (v - moyenne)**2
return math.sqrt(total/len(valeurs))

liste1 = [10, 11, 2, 5, 13]
liste2 = [1, 18, 20, 10]
liste3 = [10, 10, 10, 10]
ecart_type_1 = ecart_type(liste1)
ecart_type_2 = ecart_type(liste2)
ecart_type_3 = ecart_type(liste3)
print(ecart_type_1, ecart_type_2, ecart_type_3)
# Ou sans variable intermédiaire
print(ecart_type(liste1), ecart_type(liste2), ecart_type(liste3))

```

## 1.8 Exercice : fonction qui recherche le minimum dans une liste de valeurs

- Reprendre le code du chapitre sur les boucles qui permettait de trouver la valeur minimale dans une liste de nombres et le mettre dans une fonction `cherche_min`, qui prendra en paramètre une liste et qui retournera le minimum (si la liste est vide, on retournera `None`).
- Dans le programme principal, définir plusieurs listes et effectuer l'appel à la fonction.
- Valider que le resultat obtenu est bien celui attendu.

```

[ ]: def trouve_min(liste):
    if len(liste) == 0:
        return None

    val_min = liste[0]
    for elem in liste:
        if elem < val_min:
            val_min = elem
    return val_min

## PP
liste1 = [1, 2, 3, 4, 5]
liste2 = [6, 2, 4]
liste3 = [6, 4, 3]
liste4 = [8, 8, 8]
liste5 = [1]
liste6 = []

assert trouve_min(liste1) == 1
assert trouve_min(liste2) == 2
assert trouve_min(liste3) == 3
assert trouve_min(liste4) == 8
assert trouve_min(liste5) == 1
assert trouve_min(liste6) == None

```

## 1.9 Exercice : calcul de la vitesse

- Ecrire une fonction `calcul_vitesse_kmh` qui prend en argument la distance (en m) et le temps (minutes) et qui renvoie la vitesse en (en km/h).
- Dans le programme principal :
  - On demandera à l'utilisateur de saisir la distance et le temps et on fera ensuite appel à la fonction
  - On affichera ensuite le résultat à l'écran

```
[ ]: def calcul_vitesse_kmh(distance_m, temps_min):  
    if temps_min == 0:  
        return None # On évite une erreur de division par 0  
    return round((distance_m/1000) / (temps_min/60), 2)  
  
# Programme principal  
tps = int(input("Temps (min) : "))  
dist = int(input("Distance (m) : "))  
vitesse = calcul_vitesse_kmh(dist, tps)  
print(f"La vitesse est de {vitesse} km/h")
```

## 1.10 Exercice : factoriser le code (éviter la duplication de code)

On souhaite demander à l'utilisateur son prénom, son nom et la ville où il habite pour ensuite l'afficher. Avant l'affichage on contrôlera que le prénom, le nom et la ville contiennent uniquement des caractères alphabétiques.

*# Du code dupliqué en pagaille !*

```
saisie_ok = False  
while not saisie_ok:  
    prenom = input("Votre prénom :")  
    if prenom.isalpha():  
        saisie_ok = True  
    else:  
        print("Erreur de saisie")  
  
saisie_ok = False  
while not saisie_ok:  
    nom = input("Votre nom :")  
    if nom.isalpha():  
        saisie_ok = True  
    else:  
        print("Erreur de saisie")  
  
saisie_ok = False  
while not saisie_ok:  
    ville = input("Ville de résidence : ")  
    if ville.isalpha():
```

```

        saisie_ok = True
    else:
        print("Erreur de saisie")

print(f"Bonjour {prenom} {nom}, vous habitez {ville}.")

```

Cette approche fonctionne mais est mauvaise :

- Duplication de code : les mêmes portions de code sont répétées et si on voulait faire un autre contrôle (ex. au moins 2 caractères saisis) il faudrait le refaire pour prénom, nom et ville.
- Autre désavantage : le code est long (et donc moins lisible)

On va donc opter pour une approche plus **élégante** qui va factoriser le code. Ecrire une fonction `saisie_chaine(question)` qui prend en paramètre une question et renvoie la valeur saisie par l'utilisateur.

```

[ ]: def saisie_chaine(question):
    saisie_ok = False
    while not saisie_ok:
        reponse = input(f"{question} : ")
        if reponse.isalpha():
            saisie_ok = True
        else:
            print("Erreur de saisie")
    return reponse

# Programme principal
nom = saisie_chaine("Nom")
prenom = saisie_chaine("Prénom")
ville = saisie_chaine("Ville")
print(f"Bonjour {prenom} {nom}, vous habitez {ville}.")

```

```

[ ]: # Version intermédiaire

def caracteres_ok(chaine_a_tester):
    return chaine_a_tester.isalpha()

def saisie_chaine(question):
    saisie_ok = False
    while not saisie_ok:
        reponse = input(f"{question} : ")
        if caracteres_ok(reponse):
            saisie_ok = True
        else:
            print("Erreur de saisie")
    return reponse

# Programme principal
nom = saisie_chaine("Nom")

```



```

prenom = saisie_chaine("Prénom")
ville = saisie_chaine("Ville")
print(f"Bonjour {prenom} {nom}, vous habitez {ville}.")

```

```

[ ]: # Version qui accepte les "-" et les "'"

def caracteres_ok(chaine):
    for c in chaine:
        if not c.isalpha() and c not in ["'", "-"]:
            return False
    return True

def saisie_chaine(question):
    saisie_ok = False
    while not saisie_ok:
        reponse = input(f"{question} : ")
        if caracteres_ok(reponse):
            saisie_ok = True
        else:
            print("Erreur de saisie")
    return reponse

nom = saisie_chaine("Nom")
prenom = saisie_chaine("Prénom")
ville = saisie_chaine("Ville")
print(f"Bonjour {prenom} {nom}, vous habitez {ville}.")

```

```

[ ]: # Remarque 2
# Si on désire améliorer le programme et contrôler qu'au moins deux caractères
    ↪ sont saisis,
# il suffit de le rajouter dans la fonction :
def saisie_chaine(question):
    saisie_ok = False
    while not saisie_ok:
        valeur = input(f"{question} : ")
        if valeur.isalpha():
            if len(valeur) >= 2:
                return valeur
            else:
                print("Merci de saisir au moins deux caractères")
        else:
            print("Erreur de saisie")

# Programme principal
nom = saisie_chaine("Nom")
prenom = saisie_chaine("Prénom")
ville = saisie_chaine("Ville")

```

```
print(f"Bonjour {prenom} {nom}, vous habitez {ville}.")
```

### 1.11 Exercice génération de mot de passe

On souhaite créer un petit programme qui nous permettra d'avoir un mot de passe différent pour chaque site visité. Ainsi on créera une fonction qui entrée prendra l'identifiant utilisé pour se connecter sur le site ainsi que l'adresse du site. Elle renverra un mot de passe, élaboré à partir de ces deux paramètres. L'algorithme de génération du mot de passe est laissé à votre discrétion.

La structure du programme est la suivante :

```
def generer_mdp(identifiant, adresse_site):  
    # Les lignes ci-dessous élaborent une méthode savante pour générer un mot de passe  
    # à partir de l'identifiant et de l'adresse passés en paramètre  
    ...  
    return ...
```

```
# Programme principal
```

```
## Demander à l'utilisateur l'identifiant puis l'adresse du site  
## Appeler la fonction et afficher le mot de passe généré
```

```
[ ]: def generer_mdp(identifiant, adresse_site):  
    # On inverse l'ordre des caractères  
    valeur1 = identifiant[::-1]  
    valeur2 = adresse_site[::-1]  
    return valeur2 + valeur1  
  
ident = input("Identifiant sur le site ")  
adresse = input("Adresse du site")  
mdp = generer_mdp(ident, adresse)  
print(f"Mot de passe : {mdp}")
```

### 1.12 Exercice : calcul de la médiane

En théorie des probabilités et en statistiques, la médiane d'un ensemble de valeurs (échantillon, population, distribution de probabilités) est une valeur  $x$  qui permet de couper l'ensemble des valeurs en deux parties égales : mettant d'un côté une moitié des valeurs, qui sont toutes inférieures ou égales à  $x$  et de l'autre côté l'autre moitié des valeurs, qui sont toutes supérieures ou égales à  $x$ .

**Exemples** - Nombre impairs d'éléments : [2, 4, 9] -> médiane = 4 - Nombre pairs d'éléments : [2, 4, 6, 8] -> médiane = 5 ((4+6)/2)

#### Indications

- Reprendre l'exercice du tri à bulles dans le chapitre sur les listes et faire en sorte que le tri se fasse dans une fonction `tri_croissant(liste)`.
- Ecrire une fonction qui calcule la médiane en faisant appel à la fonction `tri_croissant`.

```

[5]: def tri_croissant(liste):
    """
    Effectue un tri croissant de la liste passé en paramètre
    Attention cette fonction modifie la liste originale
    @param liste: une liste de valeurs numériques
    @return None
    """
    permutation = True
    i = 0
    while permutation:
        permutation = False
        for elem in liste:
            if i < len(liste)-1:
                if liste[i] > liste[i+1]:
                    liste[i+1], liste[i] = liste[i], liste[i+1] # Echange de
↪valeurs
                    permutation = True
                i += 1
            i = 0

def mediane(liste):
    """
    Calcule la médiane d'une liste
    @param liste: une liste de valeurs numériques
    @return: la médiane
    """
    tri_croissant(liste)
    lg_liste = len(liste)
    if lg_liste == 0:
        return None
    if lg_liste == 1:
        return liste[0]

    i = int(lg_liste / 2)
    if lg_liste % 2 == 0: # Nb pair d'éléments
        return (liste[i-1] + liste[i]) / 2

    return liste[i]

# Programme principal
assert mediane([9, 2, 4]) == 4
assert mediane([6, 2, 4, 8]) == 5
assert mediane([]) == None

```

### 1.13 Exercice : la revanche des Dalton

Lucky Luke s'est endormi sur son cheval (Jolly Jumper). Les Dalton l'ont attrapé puis attaché sur les rails en gare de Guingamp. \* Écrire un programme qui affiche un tableau me permettant de connaître l'heure à laquelle Lucky Luke sera déchiqueté par le train parti de la gare Montparnasse (Paris) à 9h (il y a 404 km entre la gare Montparnasse et Guingamp). \* Le tableau prédira les différentes heures possibles pour toutes les vitesses de 200 km/h à 300 km/h, par pas de 10 km/h. \* Écrire une fonction *ecrabouille* qui reçoit la vitesse du train et qui affiche l'heure du drame \* Écrire le programme principal qui affiche le tableau demandé, à savoir :

```
À 200 km/h, Lucky Luke s'éteindra à 11h01
À 210 km/h, Lucky Luke s'éteindra à 10h55
À 220 km/h, Lucky Luke s'éteindra à 10h50
À 230 km/h, Lucky Luke s'éteindra à 10h45
À 240 km/h, Lucky Luke s'éteindra à 10h41
À 250 km/h, Lucky Luke s'éteindra à 10h37
À 260 km/h, Lucky Luke s'éteindra à 10h33
À 270 km/h, Lucky Luke s'éteindra à 10h30
À 280 km/h, Lucky Luke s'éteindra à 10h27
À 290 km/h, Lucky Luke s'éteindra à 10h24
À 300 km/h, Lucky Luke s'éteindra à 10h21
```

```
[ ]: def ecrabouille(vitesse):
    distance = 404
    heure_depart = 9
    heure = heure_depart + int(distance/vitesse)
    minutes = round((60 * distance / vitesse) % 60)
    print(f"À {vitesse} km/h, Lucky Luke s'éteindra à {heure}h{minutes:02d}")

for v in range(200, 301, 10):
    ecrabouille(v)
```

### 1.14 Exercice : deviner un nombre aléatoire

Objectif de l'exercice : tirer un nombre aléatoire entre *nb\_min* et *nb\_max* et le faire deviner à l'utilisateur. Voici quelques indications :

1. Ecrire une fonction `saisie_nombre(nb_min, nb_max)` :
  - Elle demande à l'utilisateur de saisir un nombre entre *nb\_min* et *\*\_nb\_max\**
  - Elle renvoie *None* si la valeur saisie par l'utilisateur est "f"
  - Elle contrôle que la valeur est bien numérique et comprise entre *nb\_min* et *nb\_max* :
    - Si c'est le cas elle renvoie la valeur
    - Sinon elle indique l'erreur et demande à l'utilisateur de saisir une autre valeur
2. On appellera cette fonction depuis le "programme principal"
  - On compte le nombre de tentatives
  - Si la valeur retournée par la fonction `saisie_nombre` est *None* on affiche à l'utilisateur qu'il a perdu
  - Sinon on effectue le test pour savoir si la valeur saisie est la valeur à deviner

- Si tel est le cas on lui affiche qu'il a gagné en X fois

```
[ ]: import random

def saisie_nombre(nb_min, nb_max):
    while True:
        valeur = input(f"Saisir une valeur entre {nb_min} et {nb_max} (ou 'f' ↵
        ↪pour terminer) :")
        if valeur == "f":
            return None
        if valeur.isnumeric():
            nombre = int(valeur)
            if nombre < nb_min or nombre > nb_max:
                print(f"Merci de saisir une valeur en {nb_min} et {nb_max}")
            else:
                return nombre
        else:
            print("Merci de saisir une valeur numérique")

# Programme principal

NB_MIN = 1
NB_MAX = 20
nb_tentatives = 0
nb_a_deviner = random.randint(NB_MIN, NB_MAX)

jeu_continue = True
while jeu_continue:
    valeur = saisie_nombre(NB_MIN, NB_MAX)
    nb_tentatives += 1
    if valeur == None:
        print(f"Vous avez abandonné au bout de {nb_tentatives}, le nombre à ↵
        ↪deviner était : {nb_a_deviner}")
        jeu_continue = False
    if valeur == nb_a_deviner:
        print(f"Bravo vous avez gagné en {nb_tentatives} fois !")
        jeu_continue = False
    elif valeur < nb_a_deviner:
        print("Valeur trop petite")
    else:
        print("Valeur trop grande")
```

### 1.15 Exercice : fonction d'extraction de valeurs numériques

On souhaite écrire la fonction `liste_valeurs_num` qui reçoit en paramètre une chaîne de caractères contenant des éléments quelconques (séparés par des espaces) et qui renvoie uniquement les valeurs numériques. On veillera également à ce que les valeurs ne soient pas dupliquées.

Exemples :

- "123 1 3" => [123, 1, 3]
- "123 abc 123 def" => [123]
- "abc def" => []

1) Commencer par réfléchir à l'algorithme

2) Indications techniques :

- La fonction `split(" ")` appliquée à une chaîne de caractères permet de mettre dans une liste les éléments d'une chaîne de caractères qui sont séparés par des espaces, par exemple : `"12 a 34".split(" ") -> ["12", "a", "34"]`
- La fonction `isnumeric()` appliquée à une chaîne de caractères permet de savoir si celle-ci contient une valeur numérique

```
[ ]: def liste_valeurs_num(chaine):
    liste_elem = chaine.split(" ")
    liste_res = []
    for elem in liste_elem:
        if elem.isnumeric() and int(elem) not in liste_res:
            liste_res.append(int(elem))
    return liste_res

# Programme principal
chaines_a_traiter = ["123 1 3", "123 abc 123 def", "abc def"]
for chaine in chaines_a_traiter:
    print(liste_valeurs_num(chaine))
```

## 1.16 Exercice : notes d'examen

```
notes_filles = [16, 13, 11, 12, 12.5, 16, 14.5, 15, 12, 14.5, 14, 14, 12, 10, 18, 15.5, 17, 16]
notes_garcons = [12, 11, 18, 15, 16, 20, 16, 17, 17, 14, 12.5, 15, 16, 14, 14, 14, 17, 10]
```

- Ecrire une fonction `moyenne(liste_valeurs)` qui retourne la moyenne en l'arrondissant
- Ecrire une fonction `ecart_type(liste_valeurs)` qui retourne l'écart-type en l'arrondissant
- Ecrire une fonction `affiche_valeurs(titre, moyenne, ecart_type)` qui affiche les résultats. Le titre sera soit "Général", "Filles" ou "Garçons"

Le programme aura la forme suivante :

```
import math

def calcul_moyenne(liste_valeurs):
    return ...

def calcul_ecart_type(liste_valeurs):
    return ...

def affiche_valeurs(titre, moyenne, ecart_type):
    print ...
```

```

# Programme principal
notes_filles = [16, 13, 15, 12, 12.5, 16, 17, 15, 12, 14.5, 14, 14, 12, 10, 18, 15.5, 17, 16]
notes_garcons = [12, 11, 18, 15, 16, 20, 16, 11, 12, 8, 12.5, 15, 16, 14, 14, 14, 17, 10]

## Appel des fonctions...

```

```

[ ]: import math

def calcul_moyenne(liste_valeurs):
    return round(sum(liste_valeurs) / len(liste_valeurs), 2)

def calcul_ecart_type(liste_valeurs):
    moyenne = calcul_moyenne(liste_valeurs)
    total = 0
    for v in liste_valeurs:
        total += (v - moyenne)**2
    return round(math.sqrt(total / len(liste_valeurs)), 2)

def affiche_valeurs(titre, moyenne, ecart_type):
    print(titre)
    print("\tMoyenne :", moyenne)
    print("\tEcart-type :", ecart_type)

# Programme principal
notes_filles = [16, 13, 15, 12, 12.5, 16, 17, 15, 12, 14.5, 14, 14, 12, 10, 18, ↵
↵15.5, 17, 16]
moyenne_filles = calcul_moyenne(notes_filles)
et_filles = calcul_ecart_type(notes_filles)

notes_garcons = [12, 11, 18, 15, 16, 20, 16, 11, 12, 8, 12.5, 15, 16, 14, 14, ↵
↵14, 17, 10]
moyenne_garcons = calcul_moyenne(notes_garcons)
et_garcons = calcul_ecart_type(notes_garcons)

notes = notes_filles + notes_garcons # Concaténation des deux listes

affiche_valeurs("Général", calcul_moyenne(notes), calcul_ecart_type(notes))
affiche_valeurs("Filles", moyenne_filles, et_filles)
affiche_valeurs("Garçons", moyenne_garcons, et_garcons)

if moyenne_filles >= moyenne_garcons:
    print("\nBravo mesdames")
else:
    print("\nLe masculin l'emporte")

```





# 07-cours\_python\_biost1\_cent1\_portee\_var

March 27, 2020

## 1 Portée des variables

On appelle *portée* d'une variable la "partie" du programme dans laquelle la variable est accessible.

### 1.1 Variables locales et variables globales

#### 1.1.1 Exercice : dérouler mentalement le programme suivant et expliquer ce qui se passe

```
def ma_fonction():  
    print(">>> Entrée dans ma_fonction")  
    var_locale = 2  
    print("var_locale :", var_locale)  
    print("var_globale :", var_globale)  
    print("<<< Sortie de ma_fonction")
```

```
# Programme principal  
var_globale = 12  
print("var_globale :", var_globale)  
ma_fonction()  
print("var_locale :", var_locale)
```

### 1.2 Arguments dans une fonction

#### 1.2.1 Arguments de type immuable (non mutable en anglais): string, int, float, bool, tuple...

Exercice : dérouler mentalement le programme suivant

- Est-il possible d'accéder à *valeur1* dans la fonction ?
- Est-il possible d'accéder à *valeur* dans le programme principal ?
- Pourquoi *valeur1* vaut toujours 7 après l'appel de la fonction (et non 49) ?
- Pourquoi *valeur2* vaut 49 ?

```
def au_carre(valeur):  
    valeur = valeur**2  
    return valeur
```

```
# Programme principal  
valeur1 = 7
```

```

valeur2 = au_carre(valeur1)
print("Après l'appel de la fonction :")
print(f"valeur1 : {valeur1} - valeur2 : {valeur2}")

```

**Exercice : le programme suivant ne fonctionne pas, le corriger**

```

def calcul_intensite(u_volts, r_ohms):
    if r_ohms != 0:
        i_amperes = u_volts / r_ohms
    else:
        return None

# Programme principal
u = 5
r = 300
calcul_intensite(u, r)
print(i_amperes)

```

```

In [1]: def calcul_intensite(u_volts, r_ohms):
        if r_ohms != 0:
            i_amperes = u_volts / r_ohms
            return i_amperes
        else:
            return None

        # Programme principal
        i_a = calcul_intensite(5, 300)
        print(i_a)

```

0.016666666666666666

```

In [2]: # Remarque : on pourrait penser que le code suivant pourrait
        # nous donner le resultat escompté, mais non !

```

```

def calcul_intensite(u_volts, r_ohms):
    if r_ohms != 0:
        # Est-ce qu'on fait référence à la variable globale
        # définie dans le programme principal ?
        i_amperes = u_volts / r_ohms
    else:
        return None

# Programme principal
i_amperes = None # Variable globale
calcul_intensite(5, 300)
print(i_amperes)

```

None

```
In [ ]: def calcul_intensite(u_volts, r_ohms):
        if r_ohms != 0:
            # Cette variable est locale et n'a rien à voir avec
            # la variable globale du même nom définie dans le programme principal
            i_amperes = u_volts / r_ohms
        else:
            return None

        # Programme principal
        i_amperes = None # Variable globale
        calcul_intensite(5, 300)
        print(i_amperes)
```

### 1.2.2 Récapitulatif

- Dans une fonction :
  - On peut accéder aux variables globales, mais on ne peut pas les modifier directement
  - Une variable locale définie dans une fonction sera supprimée après l'exécution de cette fonction.
- Dans le "programme principal" (code hors fonctions) :
  - On ne peut pas accéder aux variables définies dans les fonctions

### 1.2.3 Arguments de type modifiable (mutable en anglais): list, set, dict, ...

Exercice : écrire le programme suivant

```
def au_carre(liste):
    for i, valeur in enumerate(liste):
        liste[i] = valeur**2
    return liste

lst_orig = [2, 8, 12]
lst_carre = au_carre(lst_orig)
print("Liste originale (après appel de la fct)", lst_orig)
print("Liste de résultat", lst_carre)
```

- Que constate t-on pour *lst\_orig* après l'appel de la fonction ?
- Quelle est la différence avec l'exercice précédent (passage d'un entier) ?
- A la différence de l'exemple précédent, le paramètre *ma\_liste* lorsqu'il est modifié dans la fonction, il l'est au niveau **global** programme.
- **Quand on passe une liste en paramètre de fonction, il faut (en général) veiller à ne pas la modifier dans la fonction sous peine d'apparition "d'effets de bords".** Si la liste est modifiée directement dans la fonction, on perd le contenu original de la liste.

## Synthèse : comparaison de passage de paramètres entre un entier et une liste

```
def au_carre(valeur):  
    valeur = valeur**2  
    return valeur  
  
# Programme principal  
valeur1 = 7  
valeur2 = au_carre(valeur1)  
print("Après l'appel de la fonction :")  
print(f"valeur1 : {valeur1} - valeur2 : {valeur2}")
```

Après l'appel de la fonction :  
valeur1 : 7 - valeur2 : 49

```
def au_carre(liste):  
    for i, valeur in enumerate(liste):  
        liste[i] = valeur**2  
    return liste  
  
lst_orig = [2, 8, 12]  
lst_carre = au_carre(lst_orig)  
print("Liste originale (après appel de la fct)", lst_orig)  
print("Liste de résultat", lst_carre)
```

Liste originale (après appel de la fct) [4, 64, 144]  
Liste de résultat [4, 64, 144]

- Dans la première fonction, le paramètre *valeur1* passé à la fonction *au\_carre* n'est pas modifié à l'intérieur de la fonction car une copie est automatiquement effectuée dans le paramètre *valeur*.
- En revanche dans la seconde fonction, le paramètre *lst\_orig* passé à la fonction *au\_carre* est modifié à l'intérieur de la fonction. Le paramètre *liste* « pointe » simplement vers *lst\_orig*, il n'y a pas de copie implicite comme dans le premier cas.

## Comparaison

### 1.2.4 Exercice

- Réécrire le programme précédent de façon à ce que la liste originale ne soit pas modifiée

```
In [7]: def au_carre(liste):  
        # On va effectuer une copie de la liste passée en argument dans liste_c  
        liste_c = []  
        for valeur in liste:  
            liste_c.append(valeur**2)  
        return liste_c  
  
liste_orig = [2, 8, 12]  
liste_carre = au_carre(liste_orig)  
print("Liste originale (après appel de la fonction)", liste_orig)  
print("Liste de résultat", liste_carre)
```

Liste originale (après appel de la fonction) [2, 8, 12]  
Liste de résultat [4, 64, 144]

```
In [3]: # Autre solution avec instruction de copie de liste en Python  
def au_carre(liste):  
    liste_carre = liste[:] # On effectue une copie de la liste  
    for i, valeur in enumerate(liste):  
        liste_carre[i] = valeur**2  
    return liste_carre  
  
liste_orig = [2, 8, 12]  
liste_carre = au_carre(liste_orig)  
print("Liste originale (après appel de la fonction)", liste_orig)  
print("Liste de résultat", liste_carre)
```

Liste originale (après appel de la fonction) [2, 8, 12]  
Liste de résultat [4, 64, 144]

```
In [4]: # Autre solution (la plus "pythonique")
def au_carre(liste):
    return [v**2 for v in liste]

liste_orig = [2, 8, 12]
liste_carre = au_carre(liste_orig)
print("Liste originale (après appel de la fonction)", liste_orig)
print("Liste de résultat", liste_carre)
```

Liste originale (après appel de la fonction) [2, 8, 12]  
Liste de résultat [4, 64, 144]

### 1.2.5 Remarque

Pour se prémunir de la modification d'une liste dans une fonction, on peut utiliser un *tuple* (liste non modifiable) :

```
def au_carre(liste):
    # Cette fonction modifie la liste originale
    # Mais étant donné que l'argument passé depuis
    # le programme principal est un tuple, une erreur va se produire
    for i, valeur in enumerate(liste):
        liste[i] = valeur**2
    return liste

lst_orig = (2, 8, 12)
lst_carre = au_carre(lst_orig)
print("Liste originale (après appel de la fct)", lst_orig)
print("Liste de résultat", lst_carre)
```

### 1.2.6 Exercice : inverser une chaîne

- Ecrire une fonction qui prend en paramètre une chaîne et qui renvoie une chaîne avec les caractères inversés (ne pas utiliser l'indilage `::-1` pour effectuer l'inversion)
- Dans le programme principal
  - Demander un mot à l'utilisateur
  - Appeler la fonction et faire en sorte que le mot initialement saisi soit remplacé par le mot inversé
  - Afficher le mot inversé

```
In [9]: def inverse_chaine(chaine):
    chaine_inv = ""
    i = len(chaine) - 1
    while i >= 0:
        chaine_inv += chaine[i]
        i -= 1
```

```

    return chaine_inv

mot = input("Entrez un mot : ")
mot = inverse_chaine(mot)
print(mot)

```

```

Entrez un mot : toto
otot

```

### 1.2.7 Exercice : inverser une liste

- Ecrire une fonction qui prend en paramètre une liste de nombres et qui renvoie une liste avec les nombres inversés (ne pas utiliser l'indiciage `::-1` pour effectuer l'inversion)
- Dans le programme principal
  - Définir une liste de nombres
  - Appeler la fonction et faire en sorte que la liste initialement saisie soit remplacée par la liste inversée
  - Afficher la liste inversée

In [13]: *# Solution 1*

```

def inverse_liste(liste):
    i = 0
    j = len(liste) - 1
    while i < j:
        liste[i], liste[j] = liste[j], liste[i]
        j -= 1
        i += 1

lst = [1, 2, 3, 4, 5, 6]
inverse_liste(lst)
print(lst)

```

```
[6, 5, 4, 3, 2, 1]
```

## 1.3 Exercice récapitulatif 1 : modification d'une variable globale de type immuable

Le code suivant définit une variable globale *val*. Compléter le code ci-dessous de façon à ce qu'après l'appel de la fonction dans le programme principal, l'instruction `print(val)` affiche la valeur 3.

```

# Ecrire la fonction f1
# ...

```

```

# Programme principal
val = 2

```

```
# -> Appel fonction f1
# ...
print(val) # Le programme doit afficher 3
```

```
In [1]: def f1(ma_val):
        return ma_val + 1
```

```
        # Programme principal
        val = 2
        val = f1(val)
        print(val)
```

3

## 1.4 Exercice récapitulatif 2 : modification d'une variable globale de type modifiable

### 1.4.1 Exercice 2a : avec une liste

Le code suivant définit une variable globale *lst*. Compléter le code ci-dessous de façon à ce qu'après l'appel de la fonction dans le programme principal, l'instruction `print(lst)` affiche la valeur `[1, 2, 3]`.

```
# Ecrire la fonction f2
# ...

# Programme principal
lst = [1, 2]
# -> Appel fonction f2
# ...
print(lst) # Le programme doit afficher [1, 2, 3]
```

```
In [2]: def f2():
        lst.append(3)
```

```
        lst = [1, 2]
        f2()
        print(lst)
```

[1, 2, 3]

### 1.4.2 Exercice 2b : avec un dictionnaire

Le code suivant définit une variable globale *dico*. Compléter le code ci-dessous de façon à ce qu'après l'appel de la fonction dans le programme principal, l'instruction `print(lst)` affiche la valeur `{1: 'Fraises'}`.

```
# Ecrire la fonction f3
# ...
```

```
# Programme principal
dico = {1: 'Bananes'}
# -> Appel fonction f3
# ...
print(dico) # Le programme doit afficher {1: 'Fraises'}

In [3]: def f3():
        dico[1] = 'Fraises'

        dico = {1: 'Bananes'}
        f3()
        print(dico)

{1: 'Fraises'}
```



# 08-cours\_python\_biost1\_cent1\_dict

April 2, 2020

## 1 Les dictionnaires

### 1.1 Présentation

- Un dictionnaire est une structure de données permettant de mémoriser des couples : clé - valeur. Il est très utilisé en langage Python

Exemple :

```
fiche_contact = {  
    'prenom': 'Lucie',  
    'nom': 'Fer',  
    'tel': '0102030405',  
}
```

- Les valeurs associées aux clés peuvent être de différent type :

```
fiche_contact = {  
    'prenom': 'Lucie',  
    'nom': 'Fer',  
    'tel': '0102030405',  
    'age' : 90,  
    'hobbies': ['Surf', 'Rando', 'Python']  
}
```

- On accède à la valeur en spécifiant la clé entre **crochets** :

```
fiche_contact = {  
    'prenom': 'Lucie',  
    'nom': 'Fer',  
    'tel': '0102030405',  
    'age' : 90,  
    'hobbies': ['Surf', 'Rando', 'Python']  
}  
  
prenom_contact = fiche_contact['prenom']  
print(prenom_contact, type(prenom_contact))  
hobbies_contact = fiche_contact['hobbies']  
print(hobbies_contact, type(hobbies_contact))
```

Résultat :

```
Lucie <class 'str'>
['Surf', 'Rando', 'Python'] <class 'list'>
```

### 1.1.1 Exercice : “Nomenclature and Symbolism for Amino Acids and Peptides”

```
Ala, Alanine
Arg, Arginine
Asn, Asparagine
...
Val, Valine
Asx, Aspartic acid or Asparagine
Glx, Glutamine or Glutamic acid
```

- Créer un dictionnaire pour mémoriser ces données
- Afficher le dictionnaire
- Afficher le descriptif correspondant à 'Asx'

```
In [8]: amino_acids_peptides = {
        'Ala': 'Alanine',
        'Arg': 'Arginine',
        'Asn': 'Asparagine',
        'Val': 'Valine',
        'Asx': 'Aspartic acid or Asparagine',
        'Glx': 'Glutamine or Glutamic acid',
    }

    print(amino_acids_peptides)
    print('Asx', ":", amino_acids_peptides['Asx'])
    # Remarque : autre possibilité pour **l'affichage**
    print(f"Asx : {amino_acids_peptides['Asx']}")
```

```
{'Ala': 'Alanine', 'Arg': 'Arginine', 'Asn': 'Asparagine', 'Val': 'Valine', 'Asx': 'Aspartic acid or Asparagine', 'Glx': 'Glutamine or Glutamic acid'}
Asx : Aspartic acid or Asparagine
Asx : Aspartic acid or Asparagine
```

### 1.1.2 Même exercice mais en demandant à l'utilisateur de saisir une clé et lui afficher la description correspondante

- On traitera le cas où la clé saisie n'existe pas

```
In [3]: amino_acids_peptides = {
        'Ala': 'Alanine',
        'Arg': 'Arginine',
        'Asn': 'Asparagine',
        'Val': 'Valine',
        'Asx': 'Aspartic acid or Asparagine',
    }
```

```

        'Glx': 'Glutamine or Glutamic acid',
    }
    cle = input("Clé : ")
    if cle not in amino_acids_peptides:
        print(f"{cle} : clé inconnue")
    else:
        print(amino_acids_peptides[cle])

```

Clé :Val  
Valine

### 1.1.3 Exercice : PIB / habitant en euro des pays en Europe (2017) :

```

Luxembourg 107865.27
Norvège    91218.62
Suisse     76667.44
Irlande     74433.46
Danemark   61582.17

```

- Créer un dictionnaire pour mémoriser ces données
- Afficher le dictionnaire
- Afficher le PIB / hab. de la Suisse

```

In [6]: pib_hab = {
        'Luxembourg': 107865.27,
        'Norvège': 91218.62,
        'Suisse': 76667.44,
        'Irlande': 74433.46,
        'Danemark': 61582.17,
    }

    print(pib_hab)
    pib_suisse = pib_hab['Suisse']
    print(f"Suisse : {pib_suisse} ")

```

```

{'Luxembourg': 107865.27, 'Norvège': 91218.62, 'Suisse': 76667.44, 'Irlande': 74433.46, 'Danemark': 61582.17}
Suisse : 76667.44

```

### 1.1.4 Même exercice mais en demandant à l'utilisateur de saisir un pays et lui afficher le PIB / hab. correspondant

- On traitera le cas où le pays saisi n'existe pas (dans le dictionnaire)

```

In [1]: pib_hab = {
        'Luxembourg': 107865.27,
        'Norvège': 91218.62,
        'Suisse': 76667.44,

```

```

        'Irlande': 74433.46,
        'Danemark': 61582.17,
    }
    pays = input("Nom du pays : ")
    pays = pays.capitalize()
    if pays not in pib_hab:
        print("Ce pays n'existe pas dans les données")
    else:
        print(f"{pib_hab[pays]} ")

```

```

Nom du pays : irlande
74433.46

```

## 1.2 Affectation dynamique des valeurs d'un dictionnaire

```

fiche_contact = {} # initialisation
fiche_contact['prenom'] = 'Lucky' # 'prenom' est la clé, 'Lucky' la valeur
fiche_contact['nom'] = 'Luke'
print(fiche_contact)

```

- Résultat de l'instruction print: {'prenom': 'Lucky', 'nom': 'Luke'}
- Notez l'utilisation des accolades et des crochets en fonction de l'opération qu'on souhaite effectuer

### 1.2.1 Exercice : stockage d'éléments dans un dictionnaire

- Demander à l'utilisateur de rentrer un prénom, un nom et une ville
- Stocker ces valeurs dans un dictionnaire
- Afficher le dictionnaire

```

In [3]: donnees_u = {}
        prenom = input("Prénom :")
        # Création d'un élément dans donnees_u
        donnees_u['prenom'] = prenom
        nom = input("Nom :")
        # Création d'un autre élément dans donnees_u
        donnees_u['nom'] = nom
        ville = input("Ville :")
        donnees_u['ville'] = ville
        print(donnees_u)

```

```

Prénom :Marc
Nom :Schneider
Ville :Brest
{'prenom': 'Marc', 'nom': 'Schneider', 'ville': 'Brest'}

```

### 1.2.2 Exercice : manipulation de plusieurs dictionnaires

En 1982 la population des villes de Brest et Quimper était la suivante :

```
d_hab_1982 = {
    'brest': 156060,
    'quimper': 56907,
}
```

En 2016, Brest a désormais 6426 habitants de moins et Quimper 6498 de plus. - Créer un nouveau dictionnaire d\_hab\_2016 dans lequel on mettra à jour le nombre d'habitants, en utilisant les valeurs de d\_hab\_1982 - Afficher les deux dictionnaires

In [ ]: *# Remarque : attention lors de la copie des dictionnaires. Le code ci-dessous fonctionne*

```
d_hab_1982 = {
    'brest': 156060,
    'quimper': 56907,
}
d_hab_2016 = d_hab_1982
# En mettant à jour d_hab_1996 on met aussi à jour d_hab_1982
# Affecter un dictionnaire dans un autre ne crée donc pas de copie !
d_hab_2016['brest'] = d_hab_2016['brest'] - 6426
d_hab_2016['quimper'] = d_hab_2016['quimper'] + 6498
# Affichage du résultat
print("Données 1982 : ", d_hab_1982)
print("Données 2016 : ", d_hab_2016)

# Résultat :

# Données 1982 : {'brest': 149634, 'quimper': 63405}
# Données 2016 : {'brest': 149634, 'quimper': 63405}
```

### 1.2.3 Types de valeurs dans les dictionnaires

- Dans un dictionnaire, on peut mettre dans les valeurs, n'importe quel objet Python : chaîne, entier, float... mais aussi des éléments plus complexes : listes, tuples, sets. Ainsi si je souhaite mémoriser les notes des différentes promos dans un dictionnaire, je peux écrire par exemple :

```
notes = {
    'biost1': [18, 19, 14, 15],
    'cent_est1' : [15, 15, 17, 18],
}

notes_biost1 = notes['biost1']
print(notes_biost1)
print(type(notes_biost1))
```

Résultat :

```
[18, 19, 14, 15]  
<class 'list'>
```

- Inversement, je peux stocker les dictionnaires dans des suites d'éléments comme les listes, par exemple :

```
fiche_contact1 = {  
    'prenom': 'Lucky',  
    'nom': 'Luke',  
    'ville': 'Chicago'  
}  
fiche_contact2 = {  
    'prenom': 'Joe',  
    'nom': 'Dalton',  
    'ville': 'Jail City',  
}  
  
contacts = [fiche_contact1, fiche_contact2]  
for contact in contacts:  
    print(contact['nom'])
```

Résultat :

```
Luke  
Dalton
```

#### 1.2.4 Types de clés dans les dictionnaires

Les types possibles de clés sont les types “de base” de Python : str, int, float.

```
d1 = {  
    'prenom' : "John",  
    'nom' : "Wayne",  
}  
  
d2 = {  
    0 : "John",  
    1 : "Wayne",  
}  
  
# La clé est de type 'str'  
print(d1['prenom'])  
# La clé est de type 'int'  
# Rien à avoir ici avec l'index 0 d'une liste  
print(d2[0])
```

Résultat :

```
John  
John
```

### 1.3 Exercice : enregistrement de dictionnaires dans une liste

- Demander à l'utilisateur de rentrer un prénom, un nom et une ville
- Stocker ces valeurs dans un dictionnaire
- Ajouter ce dictionnaire à une liste
- Répéter l'opération tant que l'utilisateur ne saisit pas une valeur vide pour le prénom
- Afficher le contenu de la liste à la fin de la saisie

```
In [10]: liste_personnes = []
        while True:
            prenom = input("Prénom :")
            if prenom == "":
                break
            info_personne = {}
            info_personne['prenom'] = prenom
            info_personne['nom'] = input("Nom :")
            info_personne['ville'] = input("Ville :")
            liste_personnes.append(info_personne)
        print(liste_personnes)

# Que se passe t-il si on initialise *info_personne* **avant** la boucle *while* ?

Prénom :Alain
Nom :Terrieur
Ville :Quimper
Prénom :Alex
Nom :Terrieur
Ville :Brest
Prénom :
[{'prenom': 'Alain', 'nom': 'Terrieur', 'ville': 'Quimper'}, {'prenom': 'Alex', 'nom': 'Terrieur', 'ville': 'Brest'}]
```

### 1.4 Exercice : structurer des données

Soient les listes :

- lst\_etu = ['Thierry', 'Marc', 'Erwan']
- lst\_notes = [19, 18, 7]

Créer un dictionnaire pour stocker ces deux listes de façon à ce que la clé soit le prénom de l'étudiant et la valeur sa note.

```
In [2]: lst_etu = ['Thierry', 'Marc', 'Erwan']
        lst_notes = [19, 18, 7]
        d_etu = {}
        for i, etudiant in enumerate(lst_etu):
            d_etu[etudiant] = lst_notes[i]
        print(d_etu)

{'Thierry': 19, 'Marc': 18, 'Erwan': 7}
```

## 1.5 Parcourir un dictionnaire

On peut parcourir : - Les clés d'un dictionnaire - Les valeurs d'un dictionnaire - Les deux en même temps

```
ages_personnes = {
    'Enora': 12,
    'Erwan': 83,
    'Gael' : 44,
}

print("# Affichage clés")
for prenom in ages_personnes.keys():
    print(prenom)
print("# Affichage valeurs")
for age in ages_personnes.values():
    print(age)
print("# Les deux mon capitaine")
for prenom, age in ages_personnes.items():
    print(prenom, age)
```

Résultat :

```
# Affichage clés
Enora
Erwan
Gael
# Affichage valeurs
12
83
44
# Les deux mon capitaine
Enora 12
Erwan 83
Gael 44
```

### 1.5.1 Exercice

Soit le dictionnaire suivant :

```
ages_personnes = {
    'Enora': 12,
    'Erwan': 83,
    'Gael' : 44,
}
```

- Récupérer tous les âges, les mettre dans une liste et la trier (par ordre croissant)

```
In [6]: ages_personnes = {
        'Enora': 12,
```



```

        'Erwan': 83,
        'Gael' : 44,
    }
    liste_ages = []
    for age in ages_personnes.values():
        liste_ages.append(age)

    liste_ages = sorted(liste_ages)
    print(liste_ages)

```

[12, 44, 83]

## 1.6 Remarque : dictionnaire en argument de fonction

Comme pour les listes, il ne faut faire attention si on modifie le contenu d'un dictionnaire qui est passé en paramètre, sous peine "d'effets de bord" (après l'appel de la fonction dans le programme principal de dictionnaire n'aura plus sa valeur originale)

*# Rappel avec un type de base le paramètre n'est pas modifié par la fonction*

```

def ma_fonction(mon_int):
    mon_int = 2

a = 3
ma_fonction(a)
print("Après l'appel de la fonction", a)

```

Résultat :

Après l'appel de la fonction 3

En revanche avec un dict :

```

def ma_fonction(mon_dict):
    mon_dict['prenom'] = 'Guy'

d = {'prenom' : "Emile"}
ma_fonction(d)
print("Après l'appel de la fonction", d)

```

Résultat :

Après l'appel de la fonction {'prenom': 'Guy'}

## 1.7 Mise à jour d'un dictionnaire

Pour mettre à jour un dictionnaire existant avec un autre dictionnaire, on peut utiliser la méthode `update(...)`.

Exemple :

```

d_etu = {
    'Thierry' : 19,
    'Marc' : 18,
    'Erwan' : 7,
}
d_etu_f = {
    'Marie' : 20,
    'Gisèle' : 2,
}
d_etu.update(d_etu_f)
print(d_etu)

```

Résultat :

```
{'Thierry': 19, 'Marc': 18, 'Erwan': 7, 'Marie': 20, 'Gisèle': 2}
```

## 1.8 Exercice : stockage des notes des étudiants

Dans une célèbre école, on souhaite mémoriser les notes des étudiants sachant que :

- Chaque étudiant est caractérisé par un no. d'étudiant, un prénom et un nom
- Il y a deux semestres (s1 et s2)
- Il peut y avoir plusieurs notes par semestre

Proposer une structure permettant de stocker ces informations :

- Faire un exemple avec trois étudiants
- Pour les tests, dans le programme principal, demander à l'utilisateur le no. d'étudiant et le semestre et afficher ensuite les notes

Indications :

- Créer un dictionnaire *etudiants* qui contiendra tous les étudiants
- Pour chaque étudiant :
  - Créer un dictionnaire *info* pour les informations de chaque étudiant (nom, prénom)
  - Créer un dictionnaire *notes* pour les notes (s1 et s2)

```

In [1]: etudiants = {
        '1' : {
            'info' : {
                'prenom' : 'Lucky',
                'nom' : 'Luke',
            },
            'notes' : {
                's1' : [11, 13],
                's2' : [15, 19],
            }
        },
        '2' : {

```

```

        'info' : {
            'prenom' : 'Joe',
            'nom' : 'Dalton',
        },
        'notes' : {
            's1' : [10, 7],
            's2' : [6, 13],
        }
    },
    '3' : {
        'info' : {
            'prenom' : 'Jolly',
            'nom' : 'Jumper',
        },
        'notes' : {
            's1' : [14, 15],
            's2' : [16, 14],
        }
    },
}

```

```

# PP
id_etudiant = input("Id de l'étudiant :")
# etudiant et etudiants sont tous les deux de type dict
# edudiant est un "sous-dictionnaire" de etudiants
etudiant = etudiants[id_etudiant]
if etudiant is None:
    print("Etudiant inexistant")
else:
    semestre = input("Semestre :")
    notes_semestre = etudiant['notes'][semestre]
    if notes_semestre is None:
        print("Semestre inexistant")
    else:
        print(f"{etudiant['info']['prenom']} {etudiant['info']['nom']} : {notes_semestre}")

```

```

Id de l'étudiant :2
Semestre :s2
Joe Dalton : [6, 13]

```

## 1.9 Exercice : modélisation d'un échiquier

### 1.9.1 Présentation

Dans cet exercice on cherche un moyen efficace et élégant de modéliser un échiquier. Dans un jeu d'échec un plateau est composé de 64 cases. Horizontalement sont définies les lettres de a à h et verticalement les chiffres de 1 à 8.

On se propose ici d'utiliser un dictionnaire qui permettra de mémoriser quelle pièce (roi, reine, etc.) est présente à quelle case.

On peut utiliser des tuples de deux éléments comme clé de dictionnaire. Les tuples sont des séquences d'éléments (comme les listes) mais ils ne sont pas modifiables et donc ils peuvent être utilisés comme clé.

Exemple :

```
valeurs = (1, 2)
valeurs[0] # donne 1
valeurs[0] = 3 # erreur, on ne peut pas modifier
```

Pour notre exercice on pourrait avoir un dictionnaire qui aurait la forme suivante :

```
echiquier = {
    ('a', 1): 'Roi',
    ('a', 2): None,
    ...
    ('d', 5): 'Reine',
    ...
}
```

### 1.9.2 Question 1 : écrire une fonction `init_echiquier` qui initialise l'échiquier

Grâce à l'utilisation de boucles, créer dynamiquement un dictionnaire qui permettra de représenter l'ensemble des cases. Chaque case étant vide au départ, on lui affectera la valeur `None`.

```
In [ ]: def init_echiquier():
    lettres = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h')
    chiffres = (1, 2, 3, 4, 5, 6, 7, 8)
    echiquier = {}

    for lettre in lettres:
        for chiffre in chiffres:
            echiquier[(lettre, chiffre)] = None

    return echiquier

## PP
echiquier1 = init_echiquier()
print(echiquier1)
```

### 1.9.3 Question 2 : ajout de pièces dans les cases

Dans le programme principal:

- Ajouter manuellement quelques pièces dans des cases de votre choix.
- Vérifier le contenu de votre dictionnaire.

```
In [ ]: ## PP
echiquier1 = init_echiquier()
echiquier1[('a', 1)] = 'Roi'
echiquier1[('d', 5)] = 'Reine'
print(echiquier1)
```

#### 1.9.4 Question 3 : liste des cases occupées

- Ecrire une fonction `liste_cases_occupees` qui prend en paramètre un dictionnaire représentant un échiquier et qui renvoie en retour une liste représentant la liste des cases occupées.
- Dans le programme principal, afficher cette liste.

```
In [ ]: def liste_cases_occupees(echiquier):
    cases_occupees = []
    for case, piece in echiquier.items():
        if piece is not None:
            cases_occupees.append(case)

    return cases_occupees

## PP
echiquier1 = init_echiquier()
echiquier1[('a', 1)] = 'Roi'
echiquier1[('d', 5)] = 'Reine'
print(liste_cases_occupees(echiquier1))
```

