# Utilisation du terminal

> *Aide pour les commandes Unix :* ***man*** *(MANual).*

Ouvrez un terminal. Par défaut, vous êtes dans votre espace de travail (home).

1. Affichez les éléments contenus dans le répertoire principal (racine ~ = ou "répertoire home") en utilisant la commande **ls**.
2. Si le répertoire principal ne contient pas un sous-répertoire nommé 'Documents', créez-le en utilisant la commande **mkdir**.
3. Depuis 'Documents', créez un sous-répertoire nommé 'Septembre_2018'. Pour se déplacer d'un répertoire à un autre, utilisez la commande **cd**.
4. Créez deux fichiers 'tp1.py' et 'tp2.py' dans le dossier 'Septembre_2018'. Pour cela, utilisez la commande **touch**.
5. Quelle est la commande qui affiche à l'écran le nom de votre répertoire courant ?
6. Écrivez les commandes qui vous permettent d'aller dans le répertoire principal et d'afficher récursivement son contenu (sous-dossiers et fichiers).
7. Dessinez l'arborescence de tous les répertoires et fichiers depuis le répertoire principal.
8. Écrivez une commande qui déplace dans 'Documents' tous les fichiers .py du répertoire 'Septembre_2018'. Pour cela, utilisez la commande **mv**.
9. Écrivez une commande qui supprime le répertoire 'Septembre_2018'. Pour cela, utilisez la commande **rmdir**.

# Python

# Ceci est un commentaire en python.

Référence Python 3: https://docs.python.org/fr/3/

- Python, un langage interprété.
- Vous allez d'abord utiliser python en mode interactif, c'est-à-dire de manière à dialoguer avec lui directement depuis le clavier.
- Vous pouvez utiliser l'interpréteur directement comme une simple calculatrice de bureau.
- Dans la console python, entrez les expressions ci-dessous et observez le résultat :
- Pour ouvrir la console, il suffit de cliquer sur le bouton Python console en bas de votre fenêtre de Pycharm :
    - 5+10
    - 7 - 9
    - 7+3 * 5
    - (7 +3) *5
    - 20/6

- 8,7 / 5

- Notez que le séparateur décimal est toujours un point, et non une virgule!
- Les opérateurs arithmétiques : *+ (addition) / ( division) - (soustraction) * ( multiplication) % (reste de la division entière) // (division entière) ** (puissance)* # Variables et types

**Remarques préliminaires**

- Les variables sont utilisées pour stocker des valeurs afin que nous puissions nous y référer ultérieurement.
- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre. Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont interdits, à l'exception du caractère _ (souligné).
- La casse est significative (les caractères majuscules et minuscules sont distingués). Attention : Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !
- L'opération d'affectation est représentée par le signe égale *"="* exemple: **nom_de_la_variable** par la valeur **2** en Python: *nom_de_la_variable = 2*. Attention, elle ne signifie pas l'égalité au sens mathématiques.
- Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple : a = b = 8

- **print()** : permet d'afficher la valeur d'une variable. Exemple: *print(nom_de_la_variable)*
- Il est possible de connaître le type d'une variable en utilisant la fonction type : *print(type(nom_de_la_variable))*
- **int** : désigne un entier
- **float** : désigne un nombre décimal
- **str** : désigne une chaîne de caractères

**Exercice**

1. Affectez les valeurs **7.692** et **21.8** aux variables **a** et **b**. Calculez leur somme dans une variable nommée **resultat** et affichez le contenu de cette dernière.
2. On cherche à calculer la valeur d'un téléphone valant 999 dollars US en euros. Affectez cette valeur à une variable **telephone_usd**. Calculez la conversion en euros et affectez le résultat à une variable nommée **telephone_euro**. Rappel: 1 dollar = 0.86 euro lors de la création de cp TP.

3. Affectez la valeur **2** à une variable nommée ***ma_variable***. Quel est le type de cette variable ? Répétez l'opération avec les valeurs suivantes:

- 2.5
- 2 + 2.5
- 3 * 1.0
- 3.14 * 1
- 'z'
- "bonjour"
- 10 / 3
- 10 // 3

## Note : En python, une variable peut changer de type au gré des valeurs qu'elle prend.Exercice

1) Définissez deux variables : ***a*** et ***b*** ayant pour valeur respectivement 5 et 3.

2) Afficher la phrase suivante : "***a*** vaut 5 et ***b*** vaut 15, leur somme fait 8" en utilisant les valeurs effectives des variables.

**Note** : print() peut prendre une liste de choses à afficher séparées par des virgules. Exemple : `print("x vaut", x, "et y vaut", y)`

3) L'opérateur `%` entre deux nombres calcule le reste de la division du premier par le second. Utilisez cet opérateur pour savoir si « ***a*** est un multiple de ***b*** ? ».

4) Écrivez les lignes de code permettant d'échanger les valeurs de ***a*** et ***b***, en utilisant une variable temporaire tmp.

5) Échangez à nouveau les valeurs de ***a*** et ***b***, mais sans utiliser de variable temporaire.

**Note** : en python, on peut assigner plusieurs variables d'un seul coup à l'aide de l'opérateur égal : `x, y = 1, 2` met 1 dans x et 2 dans y. Dans cet exercice, on ne vous autorise pas à utiliser cette méthode (utilisez des additions et soustractions).# Manipulation des chaîne de caractères

***Préliminaires***

- Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication, exemple:

chaine = "Salut"

L'affichage de la variable *chaine* donne : *'Salut'*

chaine = chaine + " Python"

L'affichage de la variable *chaine* donne : *'Salut Python'* (notez l'espace avant Python).

chaine = chaine * 3

L'affichage de la variable *chaine* donne : *'SalutSalutSalut'*

- L'opérateur d'addition **+** permet de concaténer (assembler) deux chaînes de caractères et l'opérateur de multiplication ***** permet de dupliquer plusieurs fois une chaîne. On ne peut "ajouter" à une chaîne qu'une autre chaîne, et on ne peut multiplier une chaîne que par un entier.
- Lorsque l'on utilise `print()`, on peut afficher à la suite plusieurs valeurs de types différents (par exemple `print(1, "hello", 3.14)`. Cela ne fonctionne pas avec l'addition entre chaînes de caractères. Pour y arriver, il faut d'abord convertir les valeurs de types différent vers le type chaîne de caractères en utilisant `str(x)` où x est la valeur à convertir. Par exemple `resultat = str(1) + "hello" + str(3.14)`.

1) Créez une variable contenant "rouge", une autre contenant "ballon". Assignez à une troisième variable la concaténation de ces deux chaînes de manière à ce qu'elle contienne "ballon rouge". Affichez le résultat.

- Les chaînes de caractères peuvent être écrites soit encadrées par des guillemets simples 'abc', soit par des gillemets doubles "abc". Il faut utiliser le même type de guillemets avant et après le texte. On peut insérer le type de guillemet utilisé pour encadrer la chaîne dans celle-ci en le préfixant du caractère \ (antislash). Créez une chaîne de caractères contenant le mot *aujourd'hui* avec les deux méthodes.

2) Quelle est la différence entre ces deux instructions Python :

   x = 2018

   x = "2018"

Vous avez ici la notion de types et de typage : Chaque valeur en Python se voit associer un type, ceci précise quelles sont les opérations permises sur ces valeurs, et quel est leur sens. En Python les variables (comme x) n'ont pas de type au départ. C'est au moment de l'exécution, de l'évaluation des valeurs, que le système vérifie le type, et réagit "bien" ou "mal" vis à vis les opérations demandées. Cette propriété porte le nom de ***typage dynamique***.

3) Vérifiez ce qui se passe quand on évalue x+x dans les deux cas.

**Rappel:**

- La fonction print() : pour afficher des messages et des valeurs

a. Écrire un programme qui affecte les variables *note1*, *note2*, *note3* avec les valeurs 15.5, 12.75 et 14.25 respectivement. Ensuite, calculez la moyenne de ces notes et stockez le résultat dans une variable nommée *moyenne*.

Affichez la valeur de la variable *moyenne* précédée du message suivant : "La moyenne des trois notes est":

b. Si 15 étudiants utilisent 5 ordinateurs, combien utiliseront d'ordinateurs 90 étudiants ? Écrire un programme contenant des variables bien nommées avec les données du problème, puis calculant le résultat dans une variable et l'affichant.

c. Un brin d'ADN est constitué des bases A,T,C et G. Soit deux gènes "AACTG" et "GAATA", on souhaite céer un nouveau gène qui soit la concaténation du premier et du second, puis créer un brin d'ADN qui répète 7 fois le brin ainsi créé. Écrire un programme qui contienne une variable pour chacun de ces gènes (chaînes de caractères), puis calcule le brin d'ADN (concaténation et réplication) dans une nouvelle variable, et affiche le résultat.Les structures de contrôle

•

Les structures de contrôle sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois:

- la séquence
- la selection (voir partie 2)
- la répétition (voir parties 3 et 4)

**Séquence d'instructions** : une série d'instructions qui se suivent (chaînes de caractères, les tuples, les listes). Les instructions d'un programme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du script.

Par exemple, dans la séquence d'instructions suivantes :

```
a, b = 3, 7

a = b

b = a

print(a, b)
```

Vous obtiendrez un résultat contraire si vous intervertissez les 2eme et 3eme lignes.

## Expressions booléennes

Dans cette séquence, vous allez découvrir :

- le type booléen,
- les opérateurs de comparaison,
- les opérateurs logiques,

- l'instruction conditionnelle.

Les valeurs de vérité vrai et faux sont représentées respectivement en Python comme en anglais par **True** et **False** .

**Note** : La majuscule est indispensable!

Utilisez la console python pour exécuter les instructions suivantes:

- 3 == 4 `"Cet opérateur permet de tester l'égalité de valeur."`
- 3 != 4 `"Cet opérateur teste la différence."`
- not (3 == 4)
- 3 < 4
- (3 + 4) >= 7
- ( (3 + 4 ) == 7 ) && (3 <= (7 - 4))
- 2 < 3 or 4 >= 10

**Opérateurs de comparaison :**

< : strictement inferieur à

<= : inférieur ou égal à

> : strictement supérieur à

>= : supérieur ou égal à

== : égal à

!= : différent de

Les comparateurs {<, >, <=, >=, ==, !=} permettent de comparer deux valeurs numériques et renvoient un booléen.

**Attention** :

L'égalité s'écrit avec l'opérateur == à deux caractères. L'opérateur simple = est l'affectation ; il ne faut pas les confondre.

**Opérateurs booléens :**

- et (and)
- ou (or)
- non (not)

Pour le ***non*** logique :

- non faux = vrai,
- non vrai = faux.

Exemple :

"`(x%2==0) and (x>2)`" est une expression booléenne.

## Exercice à la console Python

a. Initialisez les variables a, b et c respectivement aux valeurs 2, 5 et 7. Déterminez si les tests suivants sont vrais ou faux en utilisant les opérateurs booléens et les opérateurs de comparaison correspondants :

1) a **égale à** b

2) (a + b) **différent de** c

3) **non** (a **égale à** c)

4) ((a + b) **supérieur ou égal à** c) **et** (a **inférieur ou égal à** (c − b))

5) ((a + c) > b ) **ou** (( a + c) < b)

b. Intialisez la variable trois à 3, et deux à 2. Vérifiez si la valeur de la variable **trois** est strictement supérieure à la valeur de la variable **deux**.

c. Ecrivez une expression booléenne pour vérifier si le nom est égal à "John" et qu'il n'a pas 23 ans.

## Conditions

Pour exprimer une condition, on a besoin d'une valeur qui soit vraie ou fausse. C'est une expression booléenne.

- S'il pleut, je prends mon parapluie.
- Si **x** est pair et que **x > 2**, alors **x** n'est pas premier.

  *"reste de **x** après division par **2** est **0** et **x > 2**"*

Cette valeur est de type bool :

```
if (x%2 == 0) and (x > 2):
    "x n'est pas premier ..."
```

L'instruction **if** (si) permet de n'exécuter un bloc d'instruction que si une condition (une expression booléenne) s'évalue à vrai.

```
if expr :
  # bloc d'instructions
  # qui n'est parcouru que
  # si "expr" est vraie
# suite des instructions
```

TRES IMPORTANT : Il y a une indentation de 4 espaces entre la ligne du **if**, et la ligne suivante.

**Syntaxe :**

**1. Instruction conditionnelle :**

if ( condition 1):

instruction n°1

instruction n°2

...

**2. Instruction conditionnelle avec alternative**

if ( condition 1):

instruction n°1

instruction n°2

...

else:

instruction n°3

...

**3. Cas multiples**

if ( condition 1):

instruction n°1

instruction n°2

...

elif ( condition 2):

instruction n°3

instruction n°4

...

else:

instruction n°5

...

**Remarques**

- Le début et fin des blocs d'instruction sont délimités par une indentation du bloc (les 4 espaces). Ceci signifie que indenter son code est donc PRIMORDIAL en python.

- Les deux points ':' sont indispensables après les conditions du if et du else.
- les parenthèses autour de l'expression sont optionnels.

## Exercice

1. Soit a et b deux variables ont respectivement pour valeur 6 et 10. Testez si ces deux variables sont égales, si oui affichez le message "*a égale à b*" sinon affichez le message " *a est différent de b*"
2. Ecrivez un programme python qui permet la saisie d'une valeur représentant une réponse au message : `print("Bonjour, comment ça va? ")` , ensuite vérifiez la réponse si l'utilisateur répond "Bien" affichez un message "super!" par exemple, si la réponse est différente de "Bien" affichez un autre message.

- **Input** : Permet la saisie d'une valeur au clavier. On peut invoquer la fonction input() en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur.

Exemple:

```
a = input('Entrer la valeur de a : ')
```

3. Écrivez un programme python qui demande à l'étudiant sa note, suite à cette entrée le programme affiche si l'étudiant a eu la moyenne ou non.

## Exercice

### place de cinema

### Rappel

- La fonction int(a) : permet la conversion de la variable a en entier
- La fonction str(b) : permet la conversion de la variable b en chaine de caractère

Ecrivez un programme qui permet la saisie de l'âge de la personne « *Quel est votre âge ?* ». Si la personne est mineure, le prix d'une place est de **7 euros**, si la personne est majeure le prix est de **12 euros**.

a. Vérifiez l'âge de la personne et affectez la variable **prix_total** par les valeurs correspondantes.

b. Demander à la personne si elle souhaite du pop corn, si oui ajouter **5 euros** au prix.

c. Afficher le prix total à payer. # Repetition :

En programmation, on appelle boucle un système d'instructions qui permet de répéter un certain nombre de fois toute une série d'opérations. Python propose deux instructions particulières pour construire des boucles : 1. l'instruction **For .. in**

2. l'instruction **While** (voir partie 4)

## Boucle for :

### Description:

- La commande **For… in …** est une instruction itérative qui répète les mêmes instructions plusieurs fois.
- En Python, la boucle **For** se présente typiquement de la manière suivante :

### Syntaxe:

```
for x in liste:
    Instruction1
    .....
    Instruction n
```

### Remarques très importantes:

- L'indentation est le principe selon lequel la disposition du code doit refléter sa structure. L'indentation est donc obligatoire au bon fonctionnement de votre programme.
- Si vous utilisez la touche Tab au lieu d'utiliser des espaces pour indenter votre program, alors vous devez toujours utiliser la touche Tab.
- Il faut taper deux fois Entrer pour sortir de la boucle.

### Exemple :

- Afficher 5 fois "Hello!" > `for x in [1,2,3,4,5]:`

  `print("Hello!")`

- Affichage après exécution :

Hello!

Hello!

Hello!

Hello!

Hello!

### - Commande range() - la commande range() génère par défaut une séquence de nombres entiers de valeurs croissantes, et différant d'une unité. - La commande range(n) désigne [0 , n[, tous les entiers de 0 à n-1. - On peut aussi utiliser range() avec deux, ou même trois arguments séparés par des virgules, afin de générer des séquences de nombres plus spécifiques :

```
range(start,stop,[step])
```

où:

1. *start* correspond à l'entier de départ.
2. *stop* à l'entier final.
3. *step* est optionnel et permet de faire des pas différents de 1.

---

Exercice 1

---

rire un programme qui calcule et affiche les cubes de 1 à 10 ( 1, 8, 21, … ) .

---

## Remarque: utiliser l'opérateur ** pour la puissance.Exercise 2

## Calculer la moyenne des nombres entre 100 et 250.Exercice 3

Ecrire un programme qui calcule la somme des nombre de la forme 1/2^i , pour i allant de 0 à 20. Exercise 4 -

Écrire un programme qui affiche successivement la suite de Fibonacci de 20 :

F(0) =0, F(1) = 1

F(n) = F (n-2)+ F (n-1) pour n > 1

Cela se calcule de manière itérative en utilisant deux variables qui mémorisent les deux derniers termes calculés.

Remarque:

1. Déclarer les variables f1 et f2 avec les valeurs 1 et 0 (f1 est F (n-1); f2 est F (n-2)).
2. Déclarer une variable fibo avec la valeur 20.
3. Itérer de 1 à fibo (utilisez la fonctions range()) pour calculer F (20).
4. Afficher la résultat final.

## Boucle For imbriquée:

### Description:

Boucle complètement inclue dans une autre, de sorte que cette dernière ne peut poursuivre son itération qu'après la sortie de la première.

### Syntaxe:

```
for x in ensemble:
   for y in ensemble:
     Instruction1
Instruction2
```

### Example:

Ecrire des boucles imbriquées pour imprimer un rectangle de 3 columns et 2 ligns .

```
num_cols = 3
num_lign = 2
for i in range(num_lign):
print('*', end=' ')
for j in range(num_cols-1):
    i*=j
    print('*', end=' ')
print('')
```

Affichage après exécution : `* * *  * * *`

## Exercise 5

Écrivez un programme qui affiche à l'écran une pyramide remplie d'étoiles, sur le modèle ci-dessous. La dernière ligne contient 10 étoiles

Remarque :

Il faut utiliser une boucle imbriquée.

```
*  * *  * * *  * * * *   * * * * *  * * * * * *  * * * * * * *  *
* * * * * * *  * * * * * * * * * *  * * * * * * * * * *
```

## Exercice 6

Ecrire un programme qui affiche tous les nombres premiers entre 20 et 60. Remarque: Un nombre premier est un nombre naturel divisible par exactement deux nombres distincts, donc par 1 et lui même.**Q3.** Écrivez un programme qui demande à l'utilisateur de : - calculer la suite des **i^3** tant que le résultat de ce calcul est inférieur à ***(1000i^2+500i+7)***. - en déduire la partie entière de la solution de l'équation : $-i^{3+1000i}2+500i+7=0$.

# Boucle While

- Tant que (while en anglais) est une expression qui permet d'exécuter un bloc d'instructions en boucle tant qu'une condition est vraie. Elle sert donc, comme le For, à répéter des instructions. Mais, à la différence du For, on a pas besoin de savoir à l'avance le nombre de fois qu'il faudra les répéter. **La syntaxe :**

```
while condition :
    instruction n°1 ;
    instruction n°2 ;
    …
```

- L'interpréteur commence par L'évaluation de la condition. Si la condition est vraie, on exécute la séquence et on recommence en première ligne où il y a '***while***'. Si la condition est fausse, on sort de la boucle et on exécute la suite du programme. par exemple

```
n=1
while  i < 5:
    n = n + 1
    print(n)
print("fin")
```

L'exécution donne :

```
`2 3 4 5 fin`
```

- Lorsque on utilise 'while', il faut s'assurer que le programme ne restera pas prisonnier d'une boucle infinie. Cette garantie peut se faire en s'assurant que la condition va finir par être fausse. Exemple de boucle infinie:

```
n = 10
while n >= 0:
    print(n)
    n = n + 1
print(fin)
```

L'exécution donne :

```
10 11 12 13 14 15 …1150 1151 ……...#"fin" ne s'affiche jamais
```

pour que ce programme s'arrête par il faut changer soit la condition du ***while*** en mettant par exemple (n<=10), soit on modifie la dernière instruction en mettant par exemple (n=n-1)

P.S: Dans certains cas une boucle infini peut être utile

***Q1. Écrire un programme qui demande un nombre à l'utilisateur, puis affiche les nombres allant de 1 jusqu'à ce nombre. Q2. Écrivez un programme qui fait la somme des nombres entiers entrés par l'utilisateur tant que ce dernier veut continuer.***

Indice : utilisez un nombre négatif comme -1 pour savoir si l'utilisateur veut s'arrêter ou non

---

OPTIONNEL

- Les boucles peuvent contenir des conditions d'arrêt, on peut par exemple avoir une condition d'arrêt qui dépend d'une valeur entrée. Pour sortir d'une boucle une fois la condition d'arrêt est satisfaite on utilise l'instruction '***break***'.

***Q2bis. Tentez de refaire la Q2 en utilisant un break.***

- On peut aussi utiliser des else avec la boucle while. Si la condition de la boucle n'est pas satisfaite on passe directement au deuxième bloc d'instruction

Exemple:

```
somme = 0
 while  somme < 3:
    print (somme, " est inférieur à 3")
    somme = somme + 1
 else:
    print (somme, " n'est pas inférieur à 3")
```

L'exécution donne :

```
0 est inférieur à 3
1 est inférieur à 3
2 est inférieur à 3
3 n'est pas inférieur à 3
```

Un des concepts les plus importants en programmation est le concept de fonction. Vous avez déjà manipulé des fonctions dans les précédent TP comme la fonction `print`.

Dans cet exercice, vous allez voir comment appeler une fonction. Une fonction s'appelle en donnant des valeurs que l'on appelle **arguments** entre parenthèses après le nom de la fonction. Si l'appel de fonction a plusieurs arguments, une virgule sépare deux arguments consécutifs.

Un appel de fonction s'écrit donc de la manière suivante :

```
nom_de_la_fonction(argument1, argument2, ...)
```

Les arguments servent à spécifier le comportement de l'appel de la fonction. Par exemple, dans le cas de `print`, les arguments seront les valeurs affichées par la fonction `print`. Les arguments étant des valeurs cela peut correspondre dans le code à :

- des littéraux c'est-à-dire des constantes. Exemples : `15`, `-199`, `12.9`, `"toto"`, `True`, …
- des variables et la valeur ser égale au contenu de la variable. Exemples : `x`, `sum`, …
- un autre appel de fonction. Exemples : `random()`, `sqrt(12)`
- des expressions, c'est-à-dire des valeurs liées par des opérateurs. Exemples : `12 + sum`, `f(12) * 234`, `12 < 24`, `2 * (x+2)`…

Par exemple, pour appeler la fonction `print` pour afficher l'entier `15` on écrit `print(15)`.

La fonction `print` peut être appelée avec un nombre d'arguments quelconque. Les arguments seront affichés dans l'ordre avec un espace les séparant et un saut de ligne à la fin. Un appel `print("La valeur est", value)` affichera donc `La valeur est 100` si `value` est une variable initialisée à `100`.

Vous pouvez remplacer le séparateur par défaut (une chaîne de caractères contenant un espace) par une autre chaîne de caractères (même une chaîne vide), grâce à l'argument `sep`. Il faut pour cela, spécifier la valeur de `sep` en écrivant `sep=valeur` entre les parenthèses de l'appel de fonction.

Exemples :

- `print("Bonjour", "tout", "le", "monde", sep ="*")` affichera `Bonjour*tout*le*monde`
- `print("Bonjour", "tout", "le", "monde", sep ="")` affichera `Bonjourtoutlemonde`

En plus, des arguments qui seront affichés par la focntion `print`, il est possible de changer la chaîne de caractère entre l'affichage de chaque argument (un espace par défaut) ainsi que la chaîne de caractère (un saut de ligne par défaut).

## Exercice

Dans cet exercice, vous devez :

1. Appeler la fonction `print` pour qu'elle affiche `Hello World!`
2. Initialiser la variable `your_name` pour qu'elle contienne un chaîne de caractère correpondant à votre nom.

3. Appeler la fonction `print` pour qu'elle affiche `Hello` suivi d'un espace suivi du contenu de la variable `your_name`.
4. Utiliser la fonction `print` pour afficher 100 fois le mot `to` sans espace entre les mots. En plus des fonctions comme `print` qui effectuent des actions comme afficher du texte, ils existent aussi des fonctions qui calculent des valeurs et **renvoient** (ou **retourne**) donc un résultat.

Un exemple d'une telle fonction est la fonction permettant de calculer la racine carrée d'une valeur. En Python (comme dans beaucoup d'autres langages), cette fonction se nomme `sqrt` qui est une abbriéviation de **square root** qui signifie en anglais la racine carrée.

Un appel de `sqrt(2)` renverra donc la valeur de la racine carrée de 2 qui est environ égal à `1.4142135623730951`.

Pour récupérer la valeur produite par l'appel d'une fonction, il est possible de la stocker dans une variable. Par exemple en écrivant `sqrt2 = sqrt(2)`, on affecte (ou assigne) la valeur (en fait une approximation) de la racine carrée de 2 à la variable `sqrt2`.

On peut aussi utiliser un appel de fonction dans une expression. Par exemple en écrivant `golden_ratio = (1 + sqrt(5))/2`, on affecte la valeur du nombre d'or à la variable `golden_ratio`.

La fonction `sqrt` n'est pas incluse de base dans Python contrairement à la fonction `print`. Elle est dans un module appelé `math` et il faut donc l'importer pour qu'elle soit disponible. La syntaxe (manière d'écrire) pour importer une fonction d'un module est la suivante :

```
from nom_du_module import nom_de_la_fonction
```

Pour importer la fonction `sqrt` du module `math` on a donc ajouter la ligne suivante dans le code :

```
from math import sqrt
```

Il est possible d'importer toutes les fonctions d'un module en mettant `*` en tant que nom de fonction. Par exemple, le code suivant permet d'importer toutes les fonctions du module `math`.

```
from math import *
```

## Exercice

Dans cet exercice, vous devez :

1. Utiliser la fonction `sqrt` pour initialiser une variable `square_root_of_three` à la valeur de la racine carrée de trois.

2. Utiliser la fonction `sqrt` pour calculer la distance euclidéenne entre le point de coordonnées (`x1`, `y1`) et le point de coordonnées (`x2`, `y2`) qui représentent deux point dans le plan. Si vous ne connaissez pas la formule permettant de calculer la distance entre deux points, il vous suffit de la rechercher sur internet par exemple au lien suivant.Dans certains cas, il peut y arriver que le code d'une fonction contienne plusieurs occurences du mot-clé `return`.

Considérons la fonction suivante nommée `find_multiple_in_interval` :

```
def find_multiple_in_interval(value, begin, end):
    for i in range(begin, end):
        if i % value == 0:
            return i
    return -1
```

Cette fonction parcoure tous les entiers compris dans l'interval [begin , end[ et si elle trouve un multiple de `value` dans l'intervalle, elle le renvoie. Dés que l'instruction `return` est exécutée, on sort de la fonction. S'il y a plusieurs multiples de `value` dans l'intervalle, seul le premier multiple est renvoyé, car le `return` arrête l'exécution de la fonction et le reste des entiers de l'intervalle n'est pas parcouru. Si à la fin de la boucle, aucun des entiers de l'intervalle étaient un multiple de `value` la valeur `-1` est renvoyé

## Exercice

Dans cet exercice, vous devez :

1. Écrire la fonction `absolute_value` ayant un paramètre `x` et qui renvoie la valeur absolue de `x`.
2. Afficher la valeur absolue de '-11'.Comme nous l'avez vu dans la deuxième question de l'exercice 1, ils existent des fonctions comme `sqrt` calculant et renvoyant (ou retournant) un résultat. Afin d'indiquer à l'ordinateur qu'une fonction doit renvoyer une valeur, il faut utiliser le mot-clé `return` suivi de la valeur que l'on souhaite retourner à l'intérieur du code de la fonction.

On rappelle que les valeurs peuvent être exprimées par :

- des littéraux c'est-à-dire des constantes. Exemples : `15`, `-199`, `12.9`, `"toto"`, `True`, …
- des variables et la valeur sera égale au contenu de la variable. Exemples : `x`, `s`, …
- un appel de fonction. Exemples : `random()`, `sqrt(12)`
- des expressions, c'est-à-dire des valeurs liées par des opérateurs. Exemples : `12 + s`, `f(12) * 234`, `12 < 24`, `2 * (x+2)`…

Par exemple le code suivant défini une fonction `product` qui calcule et renvoie le produit de deux nombres `number1` et `number2`.

```
def product(number1, number2):
    return number1 * number2
```

Une fois que la fonction `product` est définie, on peut l'appeler avec des arguments et récupérer la valeur calculée. Considérons le code suivant :

```
p1 = product(12, 10)
P2 = product(p1, 2)
print(p2)
```

1. La variable `p1` prend la valeur du produit de `12` fois `10` soit `120`
2. La variable `p2` prend la valeur du produit de la valeur de `p1` (égale à `120`) et de `2` soit `240`.
3. Le contenu de la variable `p2` est affiché et donc 240 est affiché en sortie.

## Exercice

Dans cet exercice, vous devez :

1. Définir une fonction `addition` ayant deux paramètres `number1` et `number2` et qui renvoie la somme des deux nombres `number1` et `number2`
2. Utiliser la fonction `addition` pour calculer la somme de 12 et 14 et mettre le résultat dans la variable `s`.
3. Afficher le contenu de la variable `s`.La syntaxe pour la définition d'une fonction ayant des paramètre est la suivante :

```
def nom_de_la_fonction(nom_parametre1, nom_parametre2, ...):
    ...
    bloc instructions
    ...
```

La liste des paramètres spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction.

Comme nous l'avons vu à l'exercice précédent les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments.

# Exercice

Dans cet exercice, vous devez :

1. Définir une fonction `print_n_hello(n)` qui affiche `n` fois “Hello!” avec un saut à la ligne entre chaque affichage.
2. Appeler la fonction `print_n_hello(n)` avec comme argument 10.

3. Écrire une fonction `print_line_multiplication_table(line_number)` qui affiche la ligne d'une table de multiplication, c'est-à-dire tous les multiples de l'entier `line_number` de une fois `line_number` à dix fois `line_number`. Chaque multiple devra être séparé par un caractère `"\t"` qui correspond à une tabulation. Par exemple, un appel `print_line_multiplication_table(3)` devra afficher la ligne suivante :

   ```
   3   6   9   12  15  18  21  24  27  30
   ```

4. Appeler la fonction `print_line_multiplication_table(line_number)` avec comme argument 3.

5. Écrire une fonction `print_multiplication_table()` qui affiche les lignes 1 à 10 d'une table de multiplication. Un appel `print_multiplication_table()` devra afficher le texte suivant :

   ```
   1   2   3   4   5   6   7   8   9   10
   2   4   6   8   10  12  14  16  18  20
   3   6   9   12  15  18  21  24  27  30
   4   8   12  16  20  24  28  32  36  40
   5   10  15  20  25  30  35  40  45  50
   6   12  18  24  30  36  42  48  54  60
   7   14  21  28  35  42  49  56  63  70
   8   16  24  32  40  48  56  64  72  80
   9   18  27  36  45  54  63  72  81  90
   10  20  30  40  50  60  70  80  90  100
   ```

6. Appeler la fonction `print_multiplication_table()`On vient de voir comment utiliser des fonctions qui existe déjà en Python. Vous allez maintenant définir vos propres fonctions.

La syntaxe pour la définition d'une fonction sans paramètre et donc sans valeur à passer en argument lors d'un appel (on verra dans l'exercice comment rajouter des paramètres) est la suivante :

```
def nom_de_la_fonction():
    ...
    bloc instructions
    ...
```

On peut faire quelques remarques :

- Le mot-clé `def` s'utilise de manière relativement similaire à `if` et `while`. La ligne doit se terminer par un double point suivi d'un bloc d'instructions qui doit être indenté.
- Tout comme pour les noms de variables, vous avez une liberté quasi-totale pour le nom des fonctions. Il vous faut néanmoins respecter les mêmes règles : pas d'espaces (on utilise `_` pour séparer les mots si le nom de la fonction en contient plusieurs), que des minuscules, pas de caractères spéciaux (accents, tilde, parenthèses, accolades, …)

## Exercice

Dans cet exercice, vous devez :

1. Définir une fonction `print_1000_hello` qui affiche 1000 fois “Hello!” avec un saut à la ligne entre chaque affichage.
2. Appeler la fonction `print_1000_hello` deux fois.
3. Définir une fonction `print_1_to_10000` qui affiche les entiers de 1 à 10000 avec un saut à la ligne entre chaque affichage.
4. Appeler la fonction `print_1_to_10000` trois fois. # Exercice Dans cet exercice, vous devez :

- Écrire une fonction `rectangle_area` ayant deux paramètres `height` et `length` et qui renvoie l'aire du rectangle correspondant.
- Écrire une fonction `rectangle_perimeter` ayant deux paramètres `height` et `length` et qui renvoie le périmètre du rectangle correspondant.
- Écrire une fonction `triangle_area` ayant trois paramètres `side1`, `side2` et `side3` (les longueurs des trois cotés) et qui renvoie l'aire du triangle correspondant. On utilisera pour cela la formule de Héron.
- Écrire une fonction `triangle_perimeter` ayant trois paramètres `side1`, `side2` et `side3` (les longueurs des trois cotés) et qui renvoie le périmètre du triangle correspondant.
- Écrire une fonction `disk_area` ayant un paramètre `radius` et qui renvoie l'aire du disque correspondant. Pour la valeur de pi, on utilisera `pi` qu'on a importé du module math (deuxième ligne du code).
- Écrire une fonction `disk_perimeter` ayant un paramètre `radius` et qui renvoie le périmètre du disque correspondant. # Exercice Dans cet exercice, vous devez :
- Écrire une fonction `sum_from_begin_to_end` ayant deux paramètres `begin` et `end` et qui renvoie la somme des entiers de `begin` à `end-1` compris.
- Écrire une fonction `fibonacci` ayant un paramètre `n` et qui renvoie le terme de rang `n` de la suite de Fibonacci.
- Écrire une fonction `syracuse_next_term(term)` qui calcule le terme suivant de la suite de Syracuse à partir de la valeur `term` du terme de la suite passée en paramètre. utiliser la division entière (opérateur `//`en Python) afin de n'avoir que des entiers.
- Écrire une fonction `print_syracuse_values(initial_term)` qui affiche tous les termes de la suite commençant par le terme initial `initial_term` jusqu'au premier terme égal à 1 compris.

- Appeler la fonction `print_syracuse_values` avec comme argument la valeur `15`.Lorsqu'une variable est passé comme argument d'un appel de fonction, c'est seulement sa valeur qui est transmise et pas la variable elle-même. Un appel de fonction ne peut donc pas modifié la valeur d'une variable. Vous allez pouvoir vérifier cette affirmation avec cet exercice.

Attention vous verrez par la suite des objets (les liste par exemple) dont l'état sera modifiable y compris par un appel de fonction.

## Exercice

Dans cet exercice, vous devez :

1. Créer une variable `a` ayant pour valeur `0`.
2. Définir un fonction `increment` ayant un paramètre `value`. Cette fonction devra augmenter de un la valeur de `value` et renvoyer la nouvelle valeur de `value`.
3. Appeler la fonction `increment` avec comme argument `a`.
4. Afficher la valeur de `a`.
5. Appeler la fonction `increment` avec comme argument `a` et stocker le retour de la focntion dans la variable `a`.
6. Afficher la valeur de `a`. Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction.

Les variables à l'intérieur de la fonctions sont donc différentes de celles à l'extérieur de la fonction même si elles ont le même nom.

## Exercice

Dans cet exercice, vous devez :

1. Créer une variable nommée `a` et qui a une valeur égale à `10`.
2. Définir une fonction `set_a_to_12` sans paramètre et qui initialise une variable `a` avec la valeur `12`.
3. Appeler la fonction `set_a_to_12`.
4. Afficher la valeur de la variable `a`.# Primalité d'un entier

## Le Problème:

- Un entier `n` est premier si il n'est divisible par aucun entier `i` tel que `1<i<n`.
- Pour savoir si un nombre est premier, il suffit de tester la divisibilité de `n` par tous les entiers entre `2` et `n/2` (arrondi à l'inférieur).
  - Si aucun diviseur de `n` n'est trouvé parmi les entiers candidats, cela signifie que `n` est premier.
  - Dès lors qu'un diviseur `n` est trouvé parmi les candidats, alors le nombre n'est pas premier.

Par exemple: - `11` n'est divisible par aucun des nombres `2`, `3`, `4`, et `5`, donc il est premier. - `18` est divisible par `3`, donc il n'est pas premier.

## Quelques outils utiles:

- `a` est divisible par `b` lorsque le reste de la division entière de `a` par `b` vaut `0`. En python, utiliser l'opérateur `%`.
- Pour effectuer une division entière, autrement dit la division arrondie à l'entier inférieur, il faut utiliser l'opérateur `//`.
- Pour énumérer l'ensemble des candidats possibles de diviseurs, vous utiliserez la fonction `range(x,y)` qui énumère tous les nombres de `[x,y[`. Attention au fait que `y` n'est pas énuméré. Si vous en avez besoin, appelez range(x,y+1).

## Exercice:

Dans cet exercice, vous devez :

- 1/ Écrire une fonction `is_prime` ayant un paramètre `n`, qui renvoie `True` lorsque `n` est premier et `False` sinon.
- 2/ Écrire une fonction `print_prime` ayant un paramètre `n`, faisant appel à la fonction `is_prime` et qui affiche soit:
  - une chaîne de la forme “Le nombre 5 est premier.” pour les nombres premiers.
  - une chaîne de la forme “Le nombre 6 n'est pas premier.” sinon.
- 3/ Appeler la fonction `print_prime` avec comme argument la valeur `4`.
- 4/ Appeler la fonction `print_prime` avec comme argument la valeur `7`.
- 5/ Appeler la fonction `print_prime` avec comme argument la valeur `15`.
- 6/ Appeler la fonction `print_prime` avec comme argument la valeur `101`.

- 7/ Écrire une fonction `print_all_prime` ayant un paramètre `n`, qui affiche la suite de tous les entiers premiers entre `2` et `n`. Par exemple, `print_all_prime(11)` affichera:

  ```
  2
  3
  5
  7
  11
  ```

- 8/ Appeler `print_all_prime` avec comme argument la valeur `101`. # Décomposition en facteurs premiers des entiers naturels

## Le Problème:

- La décomposition en facteurs premiers d'un entier `n` est la liste des nombres premiers dont le produit vaut `n` (par exemple: `20=2*2*5`)
- On veut ici écrire une fonction 'decompose' affichant les différents facteurs le composant, dans l'ordre croissant.
- Pour cela, on va avoir besoin d'une variable `i`, qui au départ vaudra 2. on va tester si `i` divise `n`. Tant que l'on n'a pas trouvé tous les diviseurs (`n` n'est pas égal à `1`)
  - Si `i` divise `n`, cela signifie que `i` est un diviseur de `n`. Il faut alors:
    - Afficher `i`
    - Diviser `n` par `i`
    - Ne pas modifier `i` (sinon on va oublier de vérifier si `i` n'apparaitrait pas plusieurs fois dans la décomposition)
  - Si `i` ne divise pas `n`, on ajoute `1` à la valeur de `i`

## Exercice:

Dans cet exercice, vous devez :

- 1/ Écrire une fonction `decompose` ayant un paramètre `n`, qui affiche une première ligne du type "L'entier 12 se décompose en:", suivi de la liste des facteurs premiers de `n`, un par ligne et du plus petit au plus grand.
  Ainsi, `decompose(12)` doit afficher sur quatre lignes:

  ```
  L'entier 12 se décompose en:
  2
  2
  3
  ```

- 2/ Appeler la fonction `decompose` avec comme argument la valeur `6`.
- 3/ Appeler la fonction `decompose` avec comme argument la valeur `19`.

- 4/ Appeler la fonction `decompose` avec comme argument la valeur `90`. # Calcul du PGCD par l'Algorithme d'Euclide

## Le Problème:

- Le pgcd de deux entiers `a` et `b` est le plus grand entier `n` qui divise `a` et `b`.
- L'algorithme d'Euclide permet de le calculer, selon le principe suivant:
  - Si `a` est divisible par `b`, le pgcd de `a` et `b` vaut `b`.
  - Sinon, soit `r` le reste de la division entière de `a` par `b`
  - Le pgcd de `a` et `b` est égal au pgcd de `b` et `r`

Par exemple, le calcul du PGCD de `90` et de `35` passe par les étapes suivantes:

```
a  = b  x q + r
90 = 35 x 2 + 20
35 = 20 x 1 + 15
20 = 15 x 1 + 5
15 = 5  x 3 + 0
```

Le PGCD de `90` et `35` est donc `5`.

## Exercice

Dans cet exercice, vous devez :

- 1/ Écrire une fonction `euclide` ayant deux paramètres `a` et `b`, qui affiche les différentes étapes du calcul du pgcd de `a` et de `b`.

  L'affichage sera composé d'une première ligne donnant les arguments de la fonction, de la suite des équations calculés par l'Algorithme, et d'une ligne finale donnant le PGCD.

  Par exemple, `euclide(90,35)` affichera:

  ```
  Calcul du PGCD de 90 et de 35 :
  90 = 35 x 2 + 20
  35 = 20 x 1 + 15
  20 = 15 x 1 + 5
  15 = 5 x 3 + 0
  Le PGCD est donc 5
  ```

- 2/ Appeler la fonction `euclide` avec comme arguments `18` et `12`.
- 3/ Appeler la fonction `euclide` avec comme arguments `7` et `5`. # Exponentiation Rapide

## Le Problème:

Le but de cet exercie est d'écrire une fonction réalisant l'opérateur d'exponentiation (x**n) en n'utilisant que les opérations + et *

- L'exponentiation d'un entier `x` par un entier `n` est l'entier xn=`x*x*...*x` (`n` fois).
- L'algorithme d'exponentiation rapide permet de calculer xn efficacement, selon le principe suivant:
  - Si `n` est égal à `0`, xn = 1
  - Si `n` est égal à `1`, xn = x
  - Si `n` est pair, alors `n` peut s'écrire `2*m` (division entière), et xn = xm * xm
  - Si `n` est impair, alors `n` peut s'écrire `2*m+1`, et xn = xm * xm * x

On utilisera une variable `resultat` initialisée à `1`. Si `n` est pair, on peut faire x=x2 et n=n/2 sans changer le résultat. Si `n` est impair, on peut multiplier `resultat` par `x`, puis faire x=x2 et n=(n-1)/2.

## Exercice:

Dans cet exercice, vous devez :

- 1/ Écrire une fonction `exponentiation_rapide` ayant deux paramètres `x` et `n`, qui retourne l'entier xn, calculé par exponentiation rapide.
- 2/ Appeler et afficher le résultat de la fonction `exponentitiation_rapide` avec comme arguments `3` et `4`.
- 3/ Appeler et afficher le résultat de la fonction `exponentitiation_rapide` avec comme arguments `2` et `10`. # Suite de Syracuse

## Le Problème:

- La suite de Syracuse depuis `x` est la suite (un) telle que:
  - u0 = x
  - Si `n` est pair, alors un+1 = un/2
  - Si `n` est impair, alors un+1 = 3*un+1
- On observe que quel que soit `x`, après un certain nombres d'étapes `n`, un=`1`, Après ce `n`, la suite alterne entre trois valeurs: `4`, `2` et `1`.
- L'objectif ici est d'afficher les termes de la suite depuis `x`, jusqu'à la première apparition de la valeur `1`.

## Exercice

Dans cet exercice, vous devez :

- 1/ Écrire une fonction `syracuse` ayant un paramètre `x`, qui affiche les entier produits par la suite de syracuse initialisée en `x`, jusqu'à atteindre `1`.
  Ainsi, `syracuse(10)` doi afficher sur sept lignes:
  `10`
  `5`
  `16`
  `8`
  `4`
  `2`
  `1`
- 2/ Appeler la fonction `syracuse` avec comme argument la valeur `4`.
- 3/ Appeler la fonction `syracuse` avec comme argument la valeur `13`.
- 4/ Appeler la fonction `syracuse` avec comme argument la valeur `25`. # Les Dictionnaires

En python, on peut aussi créer des dictionnaires (aussi appelées tableaux associatif).

### Console

Ouvrir la console python en bas à gauche, et y entrer les commandes suivantes:

```
- dico = {"Alice":1.75, "Bob":1.8, "Charlie":1.72}
- print(dico)
- type(dico)
- dico["Alice"]
- dico["Bob"]
- dico["David"]
- len(dico)
```

### Explications

- Il existe un type `dict`, qui associe des eléments à d'autres. On parle de clefs et de valeurs.
- Par exemple, le dicitionnaire `dico` associe à la clef `"Alice"` la valeur `1.72`
- On peut accéder à la valeur d'une clef avec les crochets `dico[...]`
- Si on essaye d'accéder à une clef inconnue, une Erreur est renvoyée
- `len` fonctionne aussi sur les dictionnaires, et compte le nombre de paires (clef:valeur) contenues

### Console

On continue dans la console:

```
- dico = {}
- dico["Alice"] = "Boulanger"
- dico["Bob"] = "Instituteur"
- dico["Charlie"] = "Charpentier"
- print(dico)
- dico["Alice"] = "Comptable"
- del dico["Charlie"]
- dico.pop("Bob")
- print(dico)
```

**Explications**

- On peut ajouter des clefs et des valeurs en utilisant `dico[clef] = valeur`
- Si `clef` est inconnue, une case l'associant à `valeur` est ajoutée au dictionnaire
- Si `clef` est déjà une clef de `dico`, l'ancienne valeur est remplacée par la nouvelle
- Comme pour les, liste, on peut supprimer des éléments avec `del`
- Les dictionnaires possèdent une méthode `.pop(clef)` permettant d'en supprimer une clef et sa valeur. Comme pour les listes, la valeur associée à l'élément supprimé est renvoyé en résultat.

## Exercice 1

- 1/ Créer un dictionnaire `my_dict` initialement vide.
- 2/ Associer à la clef `"Alice"` la valeur `"06 23 45 67 89"`
- 3/ Associer à la clef `"Bob"` la valeur `"06 98 76 54 32"`
- 4/ Changer la valeur d'`"Alice"` à `"07 22 44 66 88"`
- 5/ Afficher `my_dict`
- 6/ Afficher sa taille

**Explications**

Il y à trois manières d'itérer sur les éléments d'un dictionnaire avec une boulce `for`.

- On peut itérer sur les clefs :

  ```
  for clef in dico.keys():
    print(clef)
  ```

- On peut itérer sur les valeurs associées au clefs :

  ```
  for valeur in dico.values():
    print(valeur)
  ```

- Et enfin on peut itérer sur les deux en même temps :

```
for clef, valeur in dico.items():
  print(clef,":",valeur)
```

Essayez d'utiliser ces parcours en console.

## Exercice 2

- 1/ Créer une variable `parite` contenant un dictionnaire vide.
  - On souhaite y stocker des paires clef:valeur sous la forme suivante: Les clefs seront des entiers, les valeurs seront des chaînes de caractères en indiquant la parité (`"pair"` ou `"impair"`).
- 2/ Y ajouter les clefs de 10 à 20, avec leur valeurs associées.
- 3/ Afficher le contenu de `parite`, en affichant en console une ligne `"L'entier <i> est <pair/impair>"` par entrée dans le dictionnaire. # Les Listes

En python, on peut créer des listes (aussi appelées tableaux), par exemple la liste `[12,7,26]` contiens 3 valeurs: les entiers `12`, `7` et `26`.

### Console

Ouvrir la console python en bas à gauche, et y entrer les commandes suivantes:

- `l = [12, 7, 26]`
- `print(l)`
- `type(l)`
- `l[0]`
- `l[1]`
- `print(l)`
- `len(l)`

### Explications

- Il existe un type `list`, composé de plusieurs 'cases', chacune contenant une valeur
- Comme pour les tuples, `l[i]` renvoie la valeur contenue dans la i-ème case de la liste `l`, et la numérotation des cases commence à `0`
- On parle d'indice et pas de numéro de case, ainsi, la première case est d'indice `0`, la deuxième d'indice `1`, la dernière d'indice `n-1`
- La fonction `len` fonctionne aussi sur les listes

### Console

On continue dans la console:

```
- l = [12, 7, 26]
- print(l[1])
- l[1] = 77
- print(l)
```

### Explications

- On peut modifier les valeurs contenues dans les cases d'une liste, en utilisant l'assignation (le `=`)
- `l[a] = b` replace ainsi la valeur dans la case d'indice `a` par `b`
- Cela ne fonctionne pas avec les tuples : les tuples ne sont pas modifiables une fois créés

### Console

On continue dans la console:

```
- l = [12]
- l.append(1)
- print(l)
- l.append(42)
- print(l)
- print("taille de l:",len(l))
- l.pop()
- print(l)
- elem = l.pop()
- print(elem)
- print(l)
```

### Explications

- La taille d'une liste peut être modifiée, ainsi, on peut lui ajouter et lui enlever des cases
- append permet d'ajouter un élément en bout de liste
- pour l'utiliser on fait nom_de_la_liste.append(element_a_inserer)
- append n'est pas une fonction comme celles que vous avez vu jusqu'à présent, elle est appelée depuis une variable liste (c'est ce que le variable.append signifie) et agit sur celle-ci, on dit que append est une méthode des listes
- Il existe d'autres méthodes, comme 'pop'

- faire nom_de_la_liste.pop() à deux effet: il supprime la dernière case de la liste, et return la valeur contenue dans la case supprimée

**Console**

On continue dans la console:

```
- l = []
- print(l)
- l = [10, 20, 30, 40]
- l.insert(0,1)
- print(l)
- l.insert(2,25)
- print(l)
- l.pop(2)
- print(l)
- del l[0]
- print(l)
```

**Explications**

- On peut créer une liste vide, qui ne contiens aucune case avec `[]`
- On peut insérer un élément en milieu de liste avec .insert, qui prends en arguments le numéro de case où insérer (son indice), et la valeur à insérer
- Si 'l' est une variable liste de taille `n`, insérer une valeur `v` dans la case d'indice `0` (en faisant `l.insert(0,v)`) ajoute la nouvelle case devant la liste
- Insérer en indice `1` insère entre les cases d'indice `0` et `1`, et ainsi de suite
- Insérer 'v' en case d'indice `n` a le même effet que faire `l.append(v)`
- On a vu précédemment que `l.pop()` supprime la dernière case de la liste et en renvoie la valeur
- On peut aussi supprimer des case en milieu de liste, en précisant quel indice de case supprimer (`l.pop(2)` supprime la case d'indice `2`, donc la troisième case puisqu'on commence à `0`)
- `del l[i]` permet aussi de supprimer la case d'indice `i`. Contrairement à `.pop`, l'élément supprimé n'est pas renvoyé.

**Console**

On continue dans la console:

```
- l1 = [1, 2]
- l2 = [3, 4, 5]
- l3 = l1 + l2
```

- `print(l1,l2,l3)`
- `l1.extend(l2)`
- `print(l1,l2)`

**Explications**

- Il y a plusieurs manières de fusionner des listes
- Utiliser l'addition permet de le faire en créant une nouvelle liste, sans modifier les listes que l'ont fusionne
- La méthode `.extend` permet d'étendre une liste en y ajoutant une autre, cela reviens à itérer le `.append`

## Exercice 1:

- 1/ Créer une variable `ma_liste` contenant les quatre valeurs `11`, `7`, `2` et `5`
- 2/ En utilisant `print`, afficher `ma_liste`
- 3/ Afficher la valeur contenue dans la première case de `ma_liste`
- 4/ Afficher la valeur contenue dans la deuxième case de `ma_liste` (d'indice `1`)
- 5/ Afficher la valeur d'indice `3` dans `ma_liste`
- 6/ Mettre la valeur 1 dans la première case
- 7/ Ajouter une case en bout de liste, contenant la valeur `57`
- 8/ Ajouter une case en début de liste, contenant la valeur `-1`
- 9/ Supprimer la troisième case de la liste
- 10/ Afficher `ma_liste`
- 11/ Afficher le nombre de cases dans `ma_liste` (sa taille)

**Explications**

On a vu, lors des TP précédents, que faire :

```
for i in range(10):
    <code>
```

Exécute `<code>` 10 fois, pour i allant de 0 à 9.

Si `l` est une liste de taille 10, ses cases sont indexées de 0 à 9.

On peut donc itérer sur les indices des cases d'une liste `l`, et le code:

```
for i in range(len(l)):
  print(l[i])
```

affiche le contenu de `l`, une valeur par ligne.

On dit qu'on utilise une boucle for pour parcourir la liste `l`

On peut aussi itérer directement sur les valeurs contenues dans les cases d'une liste `l`:

```
for v in l:
  print(v)
```

Essayez d'utiliser ces deux parcours en console.

## Exercice 2

- 1/ créer une variable `ma_liste` contenant une liste vide
- 2/ Y ajouter les entiers de 1 à 30, avec une boucle for et .append
- 3/ Compter combien d'entiers pairs sont dans `ma_liste`, et mettre le résultat dans une variable `nb_pairs`
- 4/ Remplacer chaque entier pair dans `ma_liste` par la valeur 1
- 5/ Afficher `ma_liste`

Les listes peuvent être utilisées comme argument de fonction:

```
def fonction(liste):
    <code>
```

## Exercice 3

- 1/ Écrire une fonction `moyenne` qui prends en argument une liste (non vide) de nombres, et en `return` la moyenne.
- 2/ Appeler `moyenne` sur la liste `[12,18,2.5,10]` et en afficher le résultat.

### Console

On continue dans la console:

```
- l = [12, 7, 26, 2, 56]
- print(l[0])
- print(l[0:2])
- print(l[1:4])
- print(l[1:2])
- print(l[2:2])
```

### Explications

- A partir d'une liste `l`, on peut extraire une sous-liste, en utilisant `l[i:j]`
- `l[i:j]` contiens les éléments de `l`, entre les indices `i` et `j-1`
- Ainsi, `l[i:i+1]` est équivalent à `l[i]`, et `l[i:i]` est vide
- La sous-liste extraite est indépendante

**Console**

On continue dans la console:

```
- l = list(range(22,27))
- print(l[:3])
- print(l[3:])
```

**Explications**

- `list(range(...))` permet de créer facilement une liste d'entiers consécutifs
- `l[:j]` contiens les éléments de `l`, entre les indices `0` et `j-1`
- `l[i:]` contiens les éléments de `l`, entre les indices `i` et la fin de la liste

**Console**

On continue dans la console:

```
- x = list(range(5))
- y = x
- print(x,y)
- y.append(5)
- print(x,y)
- y = x.copy()
- print(x,y)
- y.append(5)
- print(x,y)
- y = x[:2]
- y[0] = 10
- print(x, y)
```

**Explications**

- Lorsque l'on fait `y = x`, la variable `y` designe la même liste que la variable `x`.
- Si l'une des deux est modifiée, les deux se retrouvent changées.
- Si on souhaite éviter ce comportement, on utilise `.copy()` qui met en `y` une copie (indépendante) de la liste qui se trouve dans `x`
- Modifier une sous-liste n'affecte pas sa liste parente, une sous-liste est donc une copie (partielle).

## Exercice 4

- 1/ Créer une liste `x` contenant les entiers de `0` à `9`
- 2/ En extraire une sous-liste `y` contenant sa deuxième moitié (les entiers de `5` à `9`) en utilisant les sous-listes
- 3/ Afficher `x` et `y` ## Exercice 5
- 1/ Ecrire une fonction `ma_fonction`, qui prend en argument une liste d'entiers positifs, et a le comportement suivant:
  - On trouve les indices correspondants a l'élément maximal de la liste (la plus grande valeur, qui peut apparaitre plusieurs fois)
  - On remplace par `0` le contenu de toutes les cases correspondantes
  - On affiche la liste obtenue sous la forme: `resultat <liste>`
- 2/ Suivre les instructions suivantes
  - Créer une liste `ma_liste`, contenant `[1, 4, 56, 3, 45, 56, -2, 7]`
  - Aficher `ma_liste`
  - Appeler `ma_fonction` avec comme argument `ma_liste`
  - Aficher `ma_liste`
  - réaffecter `ma_liste` à `[1, 3, 2]`
  - Appeler de nouveau `ma_fonction`, mais sur `ma_liste.copy()` cette fois-ci
  - Aficher `ma_liste`

**Remarque:**

On constate que si une liste est modifiée dans une fonction, elle est également modifiée en dehors de la fonction.

La méthode `.copy()` permet d'éviter ce comportement (si l'on souhaite que la liste passée en argument ne soit pas modifiée) # Les Tuples

En python, on peut créer des tuples (aussi appelées vecteurs), par exemple le tuple `(12,7)` est une paire, qui contiens 2 valeurs: les entiers `12` et `7`.

**Console**

Ouvrir la console python en bas à gauche, et y entrer les commandes suivantes:

```
- t = (12, 7, 26)
- print(t)
- type(t)
- t[0]
- t[1]
```

- `t[2]`
- `t[10]`
- `t[3]`
- `print(t)`
- `len(t)`

**Explications**

- Il existe un type `tuple`, composé de plusieurs 'cases', chacune contenant une valeur
- On peut accéder à ces valeurs avec les crochets (`t[0]`, `t[1]` ...)
- `t[i]` renvoie la valeur contenue dans la i-ème case du tuple `t`, mais attention, la numérotation des cases commence à `0`
- Si on essaye d'accéder à un numéro de case trop grand, une Erreur est renvoyée
- Pour un tuple de `n` éléments, les cases sont donc numérotées de `0` à `n-1`, et on ne peut donc pas accéder à `t[n]`
- `len` est une fonction qui prends un tuple en argument et renvoie sa taille (length en anglais)

**Console**

On continue dans la console:

- `t = (12, 2.1, "toto")`
- `print(l[0])`
- `print(l[1])`
- `print(l[2])`
- `print(l)`
- `t1 = (1, 2)`
- `t2 = (3, 4)`
- `print(t1 + t2)`

**Explications**

- Les tuples peuvent contenir plusieurs éléments de type différents, içi un `int` (entier), un `float` (nombre réel) et un `str` (chaîne de caractères)
- L'addition de deux tuples renvoie un tuple les fusionnant

## Exercice 1

- 1/ Créer un tuple `my_tuple` à deux éléments, `0` et `100`
- 2/ Afficher son deuxième élément en utilisant la fonction `print`

- 3/ Afficher sa taille avec `len`

## Exercice2

- 1/ On souhaite écrire une fonction notes_extremes, qui demande 5 notes (entre 0 et 20) à l'utilisateur, et en extrait la note la plus basse et la note la plus haute.
  - Le mot-cef `return`, vu précédement, permet à une fonction de renvoyer un résultat, on souhaite içi l'utiliser pour en renvoyer deux
  - on va donc utiliser un tuple à deux éléments!
- 2/ Observer l'exemple fourni. Le comportement est similaire, à celui de la question `1`, mais dans ces cas précis on peut ommetre les parenthèses formant le tuple:
  - `return a, b, c, ...` retourne le tuple `(a, b, c, ...)`
  - `x, y, ... = <tuple>` permet d'assigner directement aux varibles `x`, `y` … les valeurs de `<tuple>[0]`, `<tuple>[1]` … Recherche dans un tableau non trié ==========

On veut chercher une valeur dans un tableau (également appelé une liste en Python) qui contient une suite d'entiers non triés.

Dans ce cas, il n'y a pas d'algorithme plus efficace que de parcourir le tableau jusqu'à trouver la valeur chechée, puisqu'elle peut se trouver dans n'importe quelle position. Cet algorithme a une complexité O(n), ou n est la longeur du tableu, puisque dans le pire des cas il faut en parcourir la totalité pour verifier que la valeur cherchée n'y apparait pas.

On appelle ce type de recherche une « recherche linéaire ».

## Exercice 1

Écrire le code de la fonction `linear_search(x, a)`. La fonction doit retourner un entier (entre `0` et `len(a) - 1`) qui correspond à la position de l'element `x` dans le tableau `a` (ou de sa première occurrence, si `x` apparait plusieurs fois dans `a`), ou bien l'entier `-1` si `x` n'apparait pas dans `a`. Utiliser l'algorithme de recherche linéaire pour cet exercice.

## Exercice 2

Pour tester la fonction `linear_search`, afficher les résultats de la recherche des valeurs `2`, `5`, `10` et `6` dans le tableau `a1`.

### Exercice 3

Pour verifier le temps de calcul de `linear_search`, on va mesurer combien de seconds prend la fonction `search` dans le pire des cas (c'est-à-dire, quand la valeur cherchée n'apparait pas dans le tableau)

## Recherche dans un tableau trié

TODO: rappeler qu'est-ce qu'une recherche dichotomique (déjà vue en Intro à l'informatique sur les 3 campus)

### Exercice 4

TODO: écrire la fonction `binary_search`

### Exercice 5

TODO: tester la fonction `binary_search`

### Exercice 6

TODO: mesurer le temps de `binary_search` et le plotter avec le temps de `linear_search` pour comparaison This is the markdown document.

Write your task text here

You can add hints anywhere in task text. Copy all hint div block and change its content.

This is the markdown document.

Write your task text hereThis is the markdown document.

Write your task text here

You can add hints anywhere in task text. Copy all hint div block and change its content.

This is the markdown document.

Write your task text hereThis is the markdown document.

Write your task text hereThis is the markdown document.

Write your task text hereThis is the markdown document.

Write your task text here