Fonctions

- Un programme peut rapidement contenir beaucoup de lignes, le rendant difficile à lire. De plus, il est possible d'y retrouver des groupes de lignes de codes similaires à plusieurs endroits.
- Grâce aux fonctions, nous allons écrire du code plus lisible, plus compact et réutilisable. Les fonctions permettent ainsi de décomposer un problème complexe en sous-problèmes plus simples à résoudre.
- Nous avons déjà utilisé des fonctions pré-définies sous python : `print()` , `int()` , `type()` , `bin()` .
- Nous allons apprendre à en créer et les appeler pour les réutiliser.
- Deux nouvelles instructions : `def` et `return`

1. Définir et appeler une fonction

Pour définir une fonction, on utilise l'instruction `def` :

In [ ]:

```
def tablefois5():
    i=1
    while i <= 10:
        print(5*i, end=' ')
        i = i + 1
```

- Le nom de la fonction est choisi par le programmeur, ici `table5` .
- On remarque la présence des parenthèses et du double point.
- Le bloc d'instructions qui suit est indenté (c'est le corps de la fonction).
- Cette fonction affiche les 10 premiers termes de la table de multiplication par 5.

Si cela n'est pas encore fait, éxécuter le code ci-dessus. Que se passe-t-il ? Apparement Rien... Python n'a rien affiché, mais il a lu les instructions qui définissent la fonction.

Désormais on peut l'appeler, par son nom :

In [ ]:

```
#Exécuter le code ci-dessous :
tablefois5()
tablefois5()
tablefois5()
```

Exercice 1:

Ecrire ci-dessous le code d'une fonction appelée `carre10` qui affiche les carrés des entiers naturels de 1 à 10, puis l'appeler pour vérifier son fonctionnement.

In [ ]:

```
#Réponse :




#Appel
carre10()
```

2. Utiliser des paramètres

- Les fonctions que nous venons d'écrire éxécutent une suite d'instructions sans avoir besoin de réécrire ces instructions àchaque fois que l'on en a besoin.
- Dans les fonctions pré-définies que nous avons déjà utilisées, on transmet une valeur entre parenthèses lors de l'appel (un argument).

  Exemples :
  - `int(4.02)`
  - `print('coucou')`
  - `type(True)`

- Dans la définition de ces fonctions, il est prévu une variable particulière pour recevoir l'argument transmis. Cette variable s'appelle un paramètre. On lui choisit un nom et on place ce nom entre les parenthèses lors de la définition.

Exemple :

In [ ]:

```
#Exécuter le code ci-dessous
def tablefois(n):
    i=1
    while i <= 10:
        print(n*i, end=' ')
        i = i + 1
```

- La fonction `tablefois` utilise le paramètre `n` pour afficher les 10 premiers termes de la table de multiplication par `n` .
- Pour tester cette nouvelle fonction, il faut l'appeler avec un argument, c'est à dire choisir une valeur pour `n` :

Exemples : Exécuter les cellules ci-dessous

In [ ]:

```
tablefois(2)
```

In [ ]:

```
tablefois(7)
```

In [ ]:

```
tablefois(9)
```

Exercice 2 :

Modifier le code ci-dessous pour écrire le code d'une fonction appelée `carre` qui affiche les carrés des entiers naturels de 1 à n, puis éxécuter les cellules suivantes pour tester son bon fonctionnement avec des arguments différents.

In [ ]:

```
def carre10():
    i = 1
    while i <= 10 :
        print(i*i, end=' ')
        i = i + 1
```

In [ ]:

```
#Appel
carre(10)
```

In [ ]:

```
#Appel
carre(15)
```

In [ ]:

```
#Appel
carre(28)
```

Utiliser plusieurs paramètres

Il est possible de définir autant de paramètres que l'on veut. Il suffit de les ajouter entre les parenthèses, séparés par une virgule, lors de la définition de la fonction.

Exemple :

In [ ]:

```
def tablefois(n,fin):
    i=1
    while i<=fin:
        print (n*i, end=' ')
        i = i + 1
```

La fonction `tablefois` est définie avec 2 paramètres :

- `n` : la table souhaitée
- `fin` : le nombre de termes souhaités

Lors de l'appel, il faut donc indiquer 2 arguments :

In [ ]:

```
#Appel
tablefois(4,15)
```

Exercice 3 :

1. Afficher les 4 premiers termes de la table de multiplication par 15.
2. Afficher les 20 premiers termes de la table de multiplication par 13.
3. Afficher les 7 premiers termes de la table de multiplication par 7.

In [ ]:

```
#1.
```

In [ ]:

```
#2.
```

In [ ]:

```
#3.
```

Exercice 4 :

Modifier la fonction ci-dessous. Elle comportera 3 paramètres :

- `n` : la table de multiplication souhaitée
- `debut` : l'indice du premier terme à afficher
- `fin` : l'indice du dernier terme à afficher

L'appel de la fonction `tablefois(9,3,7)` devra afficher `27 36 45 54 63`

In [ ]:

```
def tablefois(n,fin):
    i=1
    while i<=fin:
        print (n*i, end=' ')
        i = i + 1
```

In [ ]:

```
#Appel
tablefois(9,3,7)
```

3. Renvoyer une valeur : instruction `return`

- Les fonctions que nous avons écrites jusqu'ici affichent quelque chose à l'écran. Mais ce qui est affiché ne peut pas être réutilisé pour d'autres calculs.
- Pour que le résultat soit réutilisable, il faut que la fonction renvoie une valeur.
- L'instruction `return` définit ce que doit être la valeur renvoyée par la fonction.

Exemple :

In [ ]:

```
#Exécuter le code ci-dessous
def cube(c):
    return c*c*c
```

Cette fonction renvoie le cube d'un nombre, qui peut être ensuite affécté à une autre variable ou utilisé dans un calcul.

Exemples à tester :

In [ ]:

```
cube(3)
```

In [ ]:

```
a = cube(5)
print(a)
```

In [ ]:

```
cube(3)+cube(4)
```

La fonction `cube()` renvoie un entier. Une fonction peut renoyer tout types de données. Reprenons la fonction `tablefois(n,debut,fin)` . Plutôt que d'afficher les résultats, nous allons renvoyer une liste qui contiendra ces résultats:

In [ ]:

```
#Exécuter le code ci-dessous
def tablefois(n,debut,fin):
    resultats = []
    i=debut
    while i<=fin:
        resultats.append(n*i)
        i = i + 1
    return resultats
```

- La variable `résultats` est une liste vide au départ (deuxième ligne)
- `resultats.append(n*i)` : on ajoute à chaque tour de boucle un nombre à la liste avec la méthode `append()`
- Le contenu de la variable `résultats` (une liste) est renvoyé après la terminaison de la boucle `while`
- Cette valeur de retour s'utilise ensuite comme une liste.

Exemples à tester :

In [ ]:

```
L = tablefois(9,3,7)
print(L)
```

In [ ]:

```
L[0]
```

In [ ]:

```
len(L)
```

In [ ]:

```
L.append(72)
print(L)
```

Exercice 5 :

- Définir la fonction `maxi(a,b)` qui prend en paramètres deux nombres a et b et qui renvoie le plus grand des deux.
- Ainsi :
  - `maxi(-4,2)` doit renvoyer 2
  - `maxi(-58,-40)` doit renvoyer -58
  - `maxi(5,5)` doit renvoyer 5

In [ ]:

```
#Réponse
```

Exemples à tester :

In [ ]:

```
maxi(-4,2)
```

In [ ]:

```
maxi(-58,-40)
```

In [ ]:

```
maxi(5,5)
```

Exercice 6 :

- Ecrire la fonction `maxiliste(L)` qui prend en paramètre une liste de nombres et qui renvoie le plus grand élément de cette liste(on pourra utiliser le résultat de l'exercice 15 de la feuille sur les types de données).
- Ainsi, sur les listes proposées en exemple ci-après:
  - `maxiliste(L1)` doit renvoyer 116
  - `maxiliste(L2)` doit renvoyer 70
  - `maxiliste(L3)` doit renvoyer -4

In [ ]:

```
#Réponse
```

Exemples à tester :

In [ ]:

```
L1 = [93, 103, 55, 89, 10, 21, 116, 7]
L2 = [-65, 61, -71, 64, -101, 70, -62, -57]
L3 = [-46, -109, -121, -4, -53, -70, -47, -25]
```

In [ ]:

```
maxiliste(L1)
```

In [ ]:

```
maxiliste(L2)
```

In [ ]:

```
maxiliste(L3)
```

Exercice 7 :

- Ecrire la fonction `miniliste(L)` qui prend en paramètre une liste de nombres et qui renvoie le plus petit élément de cette liste(on pourra bien sûr utiliser le résultat précédent).
- Ainsi, sur les listes proposées en exemple ci-avant:
  - `miniliste(L1)` doit renvoyer 7
  - `miniliste(L2)` doit renvoyer -101
  - `miniliste(L3)` doit renvoyer -121

In [ ]:

```
#Réponse
```

Exemples à tester :

In [ ]:

```
miniliste(L1)
```

In [ ]:

```
miniliste(L2)
```

In [ ]:

```
miniliste(L3)
```

4. Décomposer un problème

Un intérêt d'utiliser des fonctions est que cela permet de décomposer un problème complexe en sous-problèmes plus simples. Exemple : Nous allons écrire la fonction `moyenne(L)` qui calcule la moyenne des nombres contenus dans une liste `L`. Pour cela, nous allons d'abord créer la fonction `somme(L)` qui calcule la somme des entiers contenus dans la liste `L`, puis utiliser cette fonction dans la définition de la fonction `moyenne(L)`

Exercice 8 :

1. Compléter la fonction `somme(L)` qui prend en paramètre une liste de nombres et qui renvoie la somme des nombres contenus dans `L`.

Ainsi, avec les exemples ci-après, on vérifiera que:

- `somme(L1)` doit renvoyer 55
- `somme(L2)` doit renvoyer 0
- `somme(L3)` doit renvoyer 42

In [ ]:

```
#1.
def somme(liste):
    s = 0


    return s
```

Exemples à tester :

In [ ]:

```
L1=[1,2,3,4,5,6,7,8,9,10]
L2=[1,2,3,4,5,-5,-4,-3,-2,-1]
L3=[42,0]
```

In [ ]:

```
somme(L1)
```

In [ ]:

```
somme(L2)
```

In [ ]:

```
somme(L3)
```

1. En utilisant la fonction `somme(L)`, en déduire la définition de la fonction `moyenne(L)` qui prend en paramètre une liste de nombres et qui renvoie la moyenne des nombres contenus dans `L`.

- Ainsi, avec les exemples ci-après, on vérifiera que:
  - `moyenne(L1)` doit renvoyer 5.5
  - `moyenne(L2)` doit renvoyer 0.0
  - `moyenne(L3)` doit renvoyer 21.0

Rappel : la moyenne d'une liste de nombres est égale à la somme des nombres de la liste divisée par le nombre d'éléments de la liste.

In [ ]:

```
#2.Réponse
```

Exemples à tester :

In [ ]:

```
moyenne(L1)
```

In [ ]:

```
moyenne(L2)
```

In [ ]:

```
moyenne(L3)
```

5. Exercices

Certains de ces exercices reprennent des programmes vus précédemment. N'hésitez pas à les utiliser !

Exercice 9:

In [ ]:

```
# Exécuter le code ci-dessous
def suite(fin):
    resultats= []
    n = 1
    while n <= fin:
        resultats.append(n)
        n = n + 1
    return resultats

suite(10)
```

1. Expliquer le fonctionnement de la fonction `suite(fin)` .
2. Modifier cette fonction en ajoutant un deuxième paramètre `debut` pour qu'elle renvoie la liste des entiers consécutifs compris entre deux bornes, passées en arguments de la fonction :
    - Ainsi, `suite(1,5)` doit renvoyer `[1,2,3,4,5]`
1. Ajouter un troisième paramètre `pas` à cette fonction de telle sorte que la liste renvoyée contiennent les entiers compris entre deux bornes, espacés de la valeur de `pas` .
    - Ainsi, `suite(1,5,2)` doit renvoyer `[1,3,5]`

1. Réponse :

In [ ]:

```
#2.



#Appel
suite(1,5)
```

In [ ]:

```
#3.



#Appel
suite(1,5,2)
```

Exercice 10 :

- Ecrivez la fonction `mois(n)` qui prend en paramètre un entier `n` et qui renvoie le nom du énième mois de l'année.
- Ainsi `mois(5)` doit renvoyer `'mai'`

In [ ]:

```
#Réponse
```

In [ ]:

```
#Appel
mois(5)
```

Exercice 11 :

- Ecrire la fonction `complement(L)` qui prend en paramètre une liste composée de 0 et de 1 et qui renvoie la liste contenant les compléments à 1 des valeurs de la liste de départ.
- Ainsi, avec les exemples ci-après, on vérifiera que: :
  - `complement(L1)` doit renvoyer `[0,1,1,0,0,0,1,0,]`
  - `complement(L2)` doit renvoyer `[0,0,0,0]`
  - `complement(complement(L3))` doit renvoyer `[0,0]`

In [ ]:

```
#Réponse
```

Exemples à tester :

In [ ]:

```
L1=[1,0,0,1,1,1,0,1]
L2=[1,1,1,1]
L3=[0,0]
```

In [ ]:

```
complement(L1)
```

In [ ]:

```
complement(L2)
```

In [ ]:

```
complement(complement(L3))
```

Exercice 12:

Ecrire la fonction `bissextile(A)` qui prend en paramètre un entier `A` et qui renvoie le booléen `True` si l'année A est bissextile et `False` sinon. Tester avec les exemples ci-après.

Rappel : Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400

In [ ]:

```
#Réponse :
```

Exemples à tester :

In [ ]:

```
bissextile(2020)
```

In [ ]:

```
bissextile(2001)
```

In [ ]:

```
bissextile(1900)
```

In [ ]:

```
bissextile(2000)
```

Exercice 13 :

- Ecrire la fonction `occurence(lettre,mot)` qui prend en paramètres un caractère et une chaîne de caractères et qui renvoie le nombre de fois où l'on trouve ce caractère dans la chaîne.
- Ainsi, avec les exemples ci-après, on vérifiera que: :
  - `occurence('e','fournaise')` renvoie `1`
  - `occurence('a','mafate')` renvoie `2`
  - `occurence('i','maïdo')` renvoie `0`

In [ ]:

```
#Réponse :
```

Exemples à tester :

In [ ]:

```
occurence('e','fournaise')
```

In [ ]:

```
occurence('a','mafate')
```

In [ ]:

```
occurence('i','maïdo')
```

Exercice 14 :

- Ecrire la fonction `nombremots(phrase)` qui renvoie le nombre de mots contenus dans la chaîne de caractères `phrase` . On considère ici comme mots les ensembles de caractères inclus entre des espaces.
- Ainsi, avec les exemples ci-après, on vérifiera que: :
  - `nombremots(p1)` renvoie `7`
  - `nombremots(p2)` renvoie `2`
  - `nombremots(p3)` renvoie `1`

In [ ]:

```
#Réponse :
```

Exemples à tester :

In [ ]:

```
p1='Il est quelle heure au pôle nord'
p2='Auf wiedersehen'
p3='Jaioubliédemettredesespaces'
```

In [ ]:

```
nombremots(p1)
```

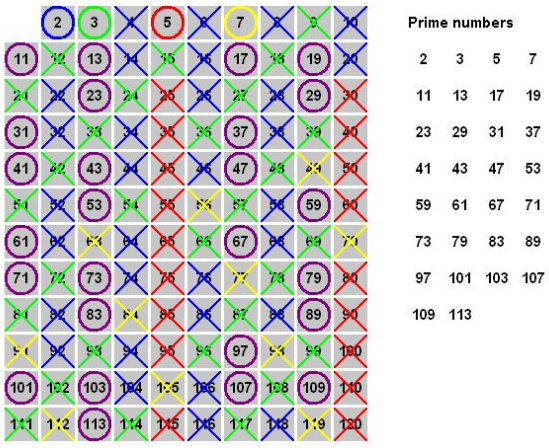
In [ ]:

```
nombremots(p2)
```

In [ ]:

```
nombremots(p3)
```

Exercice 15 : Nombres premiers



Définition :Un nombre entier naturel est dit premier s'il admet exactement 2 diviseurs qui sont 1 et lui-même.

Exemples : 1 n'est pas premier (il n'à qu'un diviseur), 2 est premier (il admet 1 et 2 comme diviseur), 4 n'est pas premier (il admet 1, 2 et 4 comme diviseurs.

Les nombres premiers sont utilisés en informatique, notamment pour sécuriser des transmissions d'informations. Il n'existe pas de formule qui permet de déterminer tous les nombres premiers. On utilise des algorithmes que l'on éxécute sur des machines toujours plus puissantes. Le plus grand nombre premier découvert à ce jour en 2018 comporte 24 862 048 chiffres !

Questions :

1. Ecrire la fonction `multiple(N,n)` qui prend en paramètres deux entiers et qui renvoie `True` si N est un multiple de n, `False` sinon.
2. Ecrire la fonction `premier(N)` qui prend en paramètre un entier naturel `N` strictement supérieur à 1 et qui renvoie `True` si N est premier, `False` sinon. On pourra réutiliser la fonction de la question précédente.
3. Ecrire la fonction `recherche(L)` qui prend en paramètres une liste d'entiers et qui renvoie la la liste des nombres premiers contenus dans la liste de départ.Vérifier que les nombres premiers compris entre 2 et 100 sont ceux de l'image ci-dessus.
4. Déterminer la liste des nombres premiers compris entre 101 et 1000.
5. 2020 n'est pas premier. Quelles sont les prochaines années "premières" ?

Remarque : Pour générer des listes d'entiers, on pourra utiliser la fonction `suite(debut,fin,pas)` de l'exercice 9.

In [ ]:

```
#1.Réponse
```

In [ ]:

```
multiple(24,2)
```

In [ ]:

```
multiple(25,3)
```

In [ ]:

```
#2.
```

In [ ]:

```
premier(13)
```

In [ ]:

```
premier(21)
```

In [ ]:

```
#3.
```

In [ ]:

```
#4.
```

In [ ]:

```
#5.
```

FIN