Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts.
Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent plus complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (Ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite).

D'autre part, il **arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme,** les fonctions permettent d'éviter de la reproduire systématiquement.

Les ***fonctions*** sont des **structures de sous-programmes**.

Nous avons déjà rencontré diverses fonctions pré-programmées (comme par exemple *print* ou *input*).
Voyons à présent comment en définir vous-mêmes de nouvelles.

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nomDeLaFonction(liste de paramètres):

    bloc d'instructions
```

Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots déjà pré-programmés (comme *input* ou *print* ...), et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis).
Comme c'est le cas pour les noms de variables, il vous est conseillé d'utiliser surtout des lettres minuscules.

La **liste de paramètres** spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction (Les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments).

1. Fonction sans paramètre :

Exemple : Taper la fonction ci-dessous sur Python :

```
>>> def table7():
...     n = 1
...     while n <11 :
...         print (n * 7)
...         n = n +1
...
```

Enregistrer ce script sous le nom table.py dans un dossier que vous nommerez test_fonction.
Que fait cette fonction ?

.....

Pour utiliser la fonction que nous venons de définir, il suffit de lancer le script (là, rien apparaît) puis l'appeler par son nom. Ainsi :

```
>>> table7()
```

Cela provoque l'affichage suivant : .....

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons.

2. Fonction qui appelle une autre fonction :

Nous pouvons également l'incorporer dans la définition d'une autre fonction, comme dans l'exemple ci-dessous :

```
import table  # importation du script contenant la fonction à utiliser
def table7triple():
      print ('La table par 7 en triple exemplaire :')
      table.table7() # appel de la fonction table7() contenue dans le script table.py
      table.table7()
      table.table7()
```

Enregistrer ce script sous le nom tripletable.py

Attention !! Pour que ce script marche, il faut l'enregistrer dans le même dossier que le script table.py dont on fait appel ! (ici dans test_fonction)

Utilisons cette nouvelle fonction, en lançant le script puis en entrant la commande :

```
>>> table7triple()
```

Une 1ère fonction peut donc appeler une 2ème fonction, qui elle-même en appelle une 3ème, etc.

Créer une nouvelle fonction vous offre l'opportunité de **donner un nom à tout un ensemble d'instructions**. De cette manière, vous pouvez **simplifier le corps principal d'un programme**, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.

Créer une nouvelle fonction **peut servir à raccourcir un programme**, par élimination des **portions de code qui se répètent**.
Par exemple, si vous devez afficher la table par 7 plusieurs fois dans un même programme, vous n'avez pas à réécrire chaque fois l'algorithme qui accomplit ce travail.

3. Fonction avec paramètre :

Nous avons défini et utilisé jusqu'ici une fonction qui affiche les termes de la table par 7.
Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un ***argument***.

Dans la définition d'une telle fonction, il faut prévoir une **variable particulière pour recevoir l'argument transmis**. Cette variable particulière s'appelle un ***paramètre***. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc.), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

Voici ce que cela donne dans le cas qui nous intéresse :

```
def table(nb):
        n = 1
        while n <11 :
        print (n * nb)
        n = n + 1
```

L'enregistrer dans le dossier test_fonction puis tester la en entrant plusieurs arguments différents.

4. Fonction avec plusieurs paramètres :

La fonction **table()** est certainement intéressante, mais elle n'affiche toujours que les 10 premiers termes de la table de multiplication, alors que nous pourrions souhaiter qu'elle en affiche d'autres.

Améliorez-la en lui ajoutant deux paramètres supplémentaires *début* et *fin ;*
Vous appellerez cette nouvelle fonction **tableMulti(nb, début, fin)** ;
Vous améliorerez également l'affichage pour que l'utilisateur sache quels calculs sont réalisés :

.....

.....

.....

.....

.....

A retenir :

Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.

Lors de l'appel de la fonction, les arguments utilisés doivent être fournis *dans le même ordre* que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.

5. L'instruction *return :*

Une fonction va pouvoir également **renvoyer une valeur** lorsqu'elle se termine. Pour cela il faudra utiliser l'instruction ***return***.

Exemple : *Enregistrer ce script sous le nom au_cube.py*

```
def cube(w):
      return w*w*w
```

L'instruction **return** définit ce que doit être la valeur renvoyée par la fonction.

Vous pouvez alors affecter à une variable la valeur retournée par votre fonction lorsque vous l'appelez :

Lancer le script contenant la fonction cube (pur l'instant rien ne se passe) puis tapez :

```
b = cube(9)
print (b)
729
```

Remarque : Il n'est pas indispensable que la valeur renvoyée par une fonction soit affectée à une variable. Ainsi, nous aurions pu tester la fonction cube() en entrant la commande :

```
>>> print (cube(9))
```

ou encore plus simplement encore :

```
>>> cube(9)
```

Exercices :

1. Modifier la fonction **table(nb)** sur laquelle nous avons travaillé, afin qu'elle renvoie une liste (la liste des dix premiers termes de la table de multiplication choisie).

2. Définir une fonction **surfCercle(R)**. Cette fonction doit renvoyer la surface (l'aire) d'un cercle dont on lui a fourni le rayon **R** en argument.
Par exemple, l'exécution de l'instruction : **print surfCercle(2.5)** doit donner le résultat **19.635**

3. Définir une fonction **volBoite(x,y,z)** qui renvoie le volume d'une boîte parallélépipède rectangle dont on fournit les trois dimensions **x, y, z** en arguments.
Par exemple, l'exécution de l'instruction : **print volBoite(5.2, 7.7, 3.3)** doit donner le résultat : **132.13**

4. Définir une fonction **maximum(a,b,c)** qui renvoie le plus grand de 3 nombres **a, b, c** fournis en arguments.
Par exemple, l'exécution de l'instruction : **print maximum(2,5,4)** doit donner le résultat : **5**