FONCTIONS avec PYTHON :

La notion de fonction utilisée en programmation et algorithmique se rapproche de celle des mathématiques. En mathématiques, une fonction associe à un paramètre une valeur.
En informatique, le principe de fonction va ressembler à celui des mathématiques.

Nom et paramètres

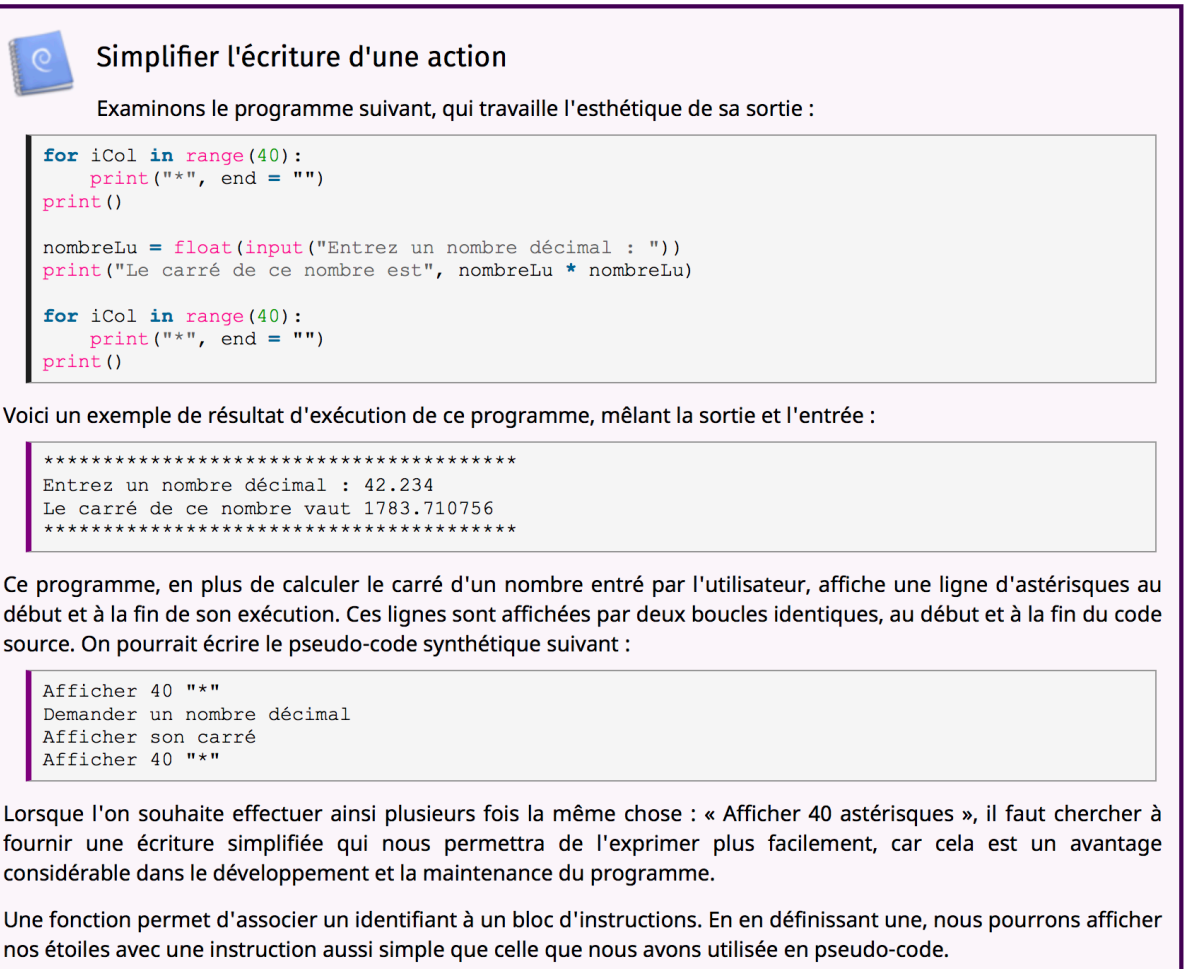
En mathématiques, on écrit $f : x \mapsto 4x + 1$. En programmation, on utilise généralement des noms longs, tout comme pour les variables, car c'est encore une occasion de mieux décrire ce que fait notre programme. Ensuite, une fonction gérera en principe un nombre fixe de paramètres (ou arguments), qui peut être nul ou multiple. Au sein de la fonction, ces paramètres seront naturellement des variables (avec un nom les décrivant si possible).

Valeur de retour

En mathématiques, une fonction associe à ses paramètres une valeur. En programmation, cela pourra être le cas : on dit alors que la fonction retourne une valeur. Néanmoins, une fonction pourra également se contenter d'exécuter des instructions.

Ecriture

En Python, la définition d'une fonction ne ressemble pas du tout à celle que l'on fait en mathématiques. En revanche, une requête à une fonction, dite « appel », en est très proche : par exemple **fonction(arg1,arg2,arg3)** pour une fonction à trois paramètres.



Simplifier l'écriture d'une action

Examinons le programme suivant, qui travaille l'esthétique de sa sortie :

```
for iCol in range(40):
    print("*", end = "")
print()

nombreLu = float(input("Entrez un nombre décimal : "))
print("Le carré de ce nombre est", nombreLu * nombreLu)

for iCol in range(40):
    print("*", end = "")
print()
```

Voici un exemple de résultat d'exécution de ce programme, mêlant la sortie et l'entrée :

```
****************************************
Entrez un nombre décimal : 42.234
Le carré de ce nombre vaut 1783.710756
****************************************
```

Ce programme, en plus de calculer le carré d'un nombre entré par l'utilisateur, affiche une ligne d'astérisques au début et à la fin de son exécution. Ces lignes sont affichées par deux boucles identiques, au début et à la fin du code source. On pourrait écrire le pseudo-code synthétique suivant :

```
Afficher 40 "*"
Demander un nombre décimal
Afficher son carré
Afficher 40 "*"
```

Lorsque l'on souhaite effectuer ainsi plusieurs fois la même chose : « Afficher 40 astérisques », il faut chercher à fournir une écriture simplifiée qui nous permettra de l'exprimer plus facilement, car cela est un avantage considérable dans le développement et la maintenance du programme.

Une fonction permet d'associer un identifiant à un bloc d'instructions. En en définissant une, nous pourrons afficher nos étoiles avec une instruction aussi simple que celle que nous avons utilisée en pseudo-code.

Définition d'une fonction

Le programme suivant définit une fonction qui affiche la ligne d'étoiles :

```
def ligneÉtoiles():
    for iCol in range(40):
        print("*", end = "")
    print()

ligneÉtoiles()
nombreLu = float(input("Entrez un nombre décimal : "))
print("Le carré de ce nombre est", nombreLu * nombreLu)
ligneÉtoiles()
```

Nous avons repris précisément le contenu du bloc répété et l'avons placé tout au début du code du programme, avec au-dessus la ligne indiquant le nom de la fonction :

```
def ligneÉtoiles():
```

Vous pouvez donc créer une fonction avec cette syntaxe en Python. Nous verrons un peu plus tard ce que quel est le rôle des parenthèses sans contenu.

La fonction peut ensuite être appelée avec une instruction telle que :

```
ligneÉtoiles()
```

Remarques

Les noms que l'on peut utiliser comme identifiants de fonctions sont les mêmes que ceux possibles pour les identifiants de variables. Comme pour le choix des noms de variables, il est important de bien choisir les noms des fonctions, pour qu'ils expriment très clairement ce que fait la fonction.

On pourrait dire qu'appeler une fonction revient à remplacer dans le programme, cet appel par le bloc d'instructions correspondant à l'identifiant de la fonction. C'est effectivement le cas ; toutefois, il faut noter que la fonction ne dispose pas des variables du contexte appelant. Comme pour n'importe quel bloc, **une fonction dispose de ses propres variables, ou des variables définies en dehors**. Nous verrons plus tard qu'il est possible de définir des variables en dehors des fonctions.

Fonctions à un paramètre

Nous venons de voir comment les fonctions permettent d'éviter de répéter exactement les mêmes instructions à plusieurs endroits d'un programme. Il arrive aussi souvent que l'on répète les mêmes instructions, mais avec de légères différences d'une version à l'autre. Supposons par exemple que nous souhaitions afficher une ligne d'étoiles au début et une ligne de tirets un peu plus loin. Les instructions utilisées dans les deux cas seront identiques à un caractère près.

On va donc rendre notre fonction paramétrable avec un argument *caractere* (et nous changeons son nom) :

```
def ligneÉtoiles(caractère):
    for iCol in range(40):
        print(caractère, end = "")
    print()
```

On place donc le paramètre nommé entre les parenthèses après le nom de la fonction.

Au sein de cette fonction, l'identifiant *caractere* représente une variable, qui a été initialisée à la valeur indiquée lors de l'appel.

On peut l'utiliser comme une variable donc également modifier sa valeur au cours de la fonction. Cela n'est cependant pas conseillé : en général, les arguments d'une fonction sont simplement vus comme des paramètres qui vont influencer leur déroulement, et pas comme des boîtes destinées à être modifiées. Le faire est ainsi une pratique inhabituelle qui nuit à la compréhension.

Pour appeler la fonction, on place la valeur entre les parenthèses :

```
ligneCaractères("*")
```

Dans le cas où l'on modifierait la valeur d'un paramètre, il est important de noter que cette modification n'a d'effet qu'à l'intérieur de la fonction. Si le paramètre a été passé sous la forme d'une variable au moment de l'appel, celle-ci ne sera pas modifiée, comme le montre le témoin suivant :

```
def afficheEtModifie(paramètre):
    print("Début fonction : ", parametre)
    paramètre = 68
    print("Fin fonction : ", parametre)

valeur = 42
print("Début programme :", valeur)
afficheEtModifie(valeur)
print("Fin programme :", valeur)
```

```
Début programme : 42
Début fonction : 42
Fin fonction : 68
Fin programme : 42
```

On gardera en tête que lorsque l'on passe une valeur en paramètre à une fonction, cette valeur est **copiée** dans une variable de la fonction correspondant à ce paramètre. Vous ne pouvez donc pas pour l'instant modifier la valeur d'une variable à partir d'une fonction.

Il faut cependant noter la différence entre la variable et sa valeur : si une valeur est modifiée, alors cela a aussi un impact à l'extérieur de la fonction. C'est le cas par exemple si on ajoute une case à un tableau. Nous ne proposons pas encore de cours qui est clair sur ce sujet — qui est d'ailleurs souvent assez confus dans l'enseignement de l'informatique.

Fonctions à plusieurs paramètres

Dans l'exemple de la fonction qui affiche plusieurs fois une ligne de caractères, on peut imaginer que le nombre de fois où le caractère doit être affiché n'est pas toujours identique et doit être passé en paramètre lors de l'appel. Pour cela, il suffit de séparer les paramètres par une virgule :

```
ligneCaractères("*", 40)
ligneCaractères("-", 35)
```

Pour pouvoir faire un tel appel, il faut cependant avoir défini dans la déclaration de la fonction que celle-ci prend un deuxième paramètre, cette fois un entier, puis modifier les instructions pour qu'elles utilisent la variable correspondante. Comme pour l'appel, on ajoute simplement un paramètre dans la déclaration en le séparant du premier par une virgule. On nommera par exemple ce paramètre *longueur* :

```
def ligneCaractères(caractère, longueur):
    for iCol in range(longueur):
        print(caractère, end = "")
    print()
```

Remarque : si l'on a défini que la fonction prenait deux paramètres, on ne peut plus l'appeler en n'en fournissant qu'un seul. En Python, il est toutefois possible de rendre certains paramètres facultatifs.

Valeur de retour

Les fonctions que nous avons écrites effectuaient des opérations, mais n'avaient cependant aucune influence sur la suite de l'exécution du programme. Supposons par exemple que nous ayons souvent besoin de calculer la valeur absolue d'un nombre, de cette façon :

```
if nombre < 0:
    distance = -nombre
else:
    distance = nombre
```

Nous pouvons écrire une fonction qui prend en paramètre un nombre et calcule sa valeur absolue. Il nous faut cependant un moyen pour retransmettre cette valeur au programme. Au moment de l'appel, l'idée est de considérer dans le programme, qu'une fonction appelée a une valeur que l'on peut utiliser ensuite. Cette valeur, calculée par les instructions de la fonction, peut alors être affectée à une variable. On pourrait ainsi écrire notre programme en utilisant la fonction *valeurAbsolue* :

```
distance1 = valeurAbsolue(nombre1);
distance2 = valeurAbsolue(nombre2);
```

Notre fonction *valeurAbsolue* prendrait en paramètre une valeur et *retournerait* sa valeur absolue. Voyons maintenant comment écrire une telle fonction.

Pour que la fonction puisse effectivement retourner une valeur, il faut qu'elle contienne une instruction, composée du mot-clé `return` et de la valeur que l'on veut retourner. L'instruction `return` quitte la fonction et transmet la valeur qui suit au programme appelant :

```
def valeurAbsolue(valeur):
    if valeur >= 0:
        valAbsolue = valeur
    else:
        valAbsolue = -valeur
    return valAbsolue
```

Une telle fonction est ainsi comparable à la notion de fonction mathématique : `valeurAbsolue(x)` aura pour valeur la valeur absolue du contenu de la variable *x*.

Remarques : l'instruction `return`

Il est possible de mettre plusieurs instructions `return` à l'intérieur d'une fonction. Chacune d'entre elles doit être suivie d'une valeur du format attendu. Il est également important que quoi qu'il arrive dans la fonction, elle retourne une valeur, donc exécute une instruction `return`. Notre fonction peut ainsi s'écrire plus simplement :

```
def valeurAbsolue(valeur):
    if valeur < 0:
        return -valeur
    return valeur
```

Dans le cas des fonctions qui ne retournent pas de valeur, il est également possible d'utiliser l'instruction `return`. Elle n'est alors suivie d'aucune valeur et a simplement pour effet de quitter la fonction immédiatement, lorsqu'elle est exécutée. En réalité, en Python, toute expression a une valeur, et une absence de valeur vaut `None`.

PROTOTYPE DE FONCTION :

Quand on veut utiliser une fonction, pour l'appeler, on a besoin des informations suivantes :

- son nom ;
- ses paramètres et leur type ;
- le type de la valeur de retour (s'il y en a une).

Ces informations sont communément appelées le *prototype* de la fonction. Cela correspond en fait, dans le code source, à l'en-tête de la fonction. En voici deux exemples pour vous y ramener :

```
def dessinerRectangle(ligne, colonne, motif):
def nombreDeSecondes(heures, minutes, secondes):
```

C'est ce qu'il est nécessaire de connaître pour appeler une fonction. Notez toutefois qu'en Python, l'en-tête de la fonction n'impose pas le type des paramètres et de la valeur de retour (en pratique, celle-ci respecte toutefois un certain format, qu'il faut connaître pour utiliser la fonction).

TYPES STRUCTURES :

Simples enregistrements

Lorsque l'on a beaucoup de données, il peut s'avérer intéressant de les regrouper sous un seul nom. Imaginons par exemple que l'on ait de nombreux dessins de rectangle à gérer ; on souhaiterait alors pouvoir regrouper ensemble leur nombre de lignes et de colonnes, ainsi que le caractère de remplissage.

Il faut alors définir un type : une *structure* ou *enregistrement*. Nous définissons ci-dessous un type *DessinRectangle*. Ce code peut être placé n'importe où : dans une fonction, dans une autre structure, ou à l'extérieur de tout bloc. En général, nous serons dans ce dernier cas, afin de pouvoir utiliser notre structure dans tout notre programme, sans le moindre souci.

```
class DessinRectangle:
    pass
```

Maintenant, on peut définir des variables (des « instances ») de ce type. Nous créons ci-dessous trois rectangles :

```
rectLong = DessinRectangle()
rectLong.nbLig = 3
rectLong.nbCol = 12
rectLong.motif = '>'

rectHaut = DessinRectangle()
rectHaut.nbLig = 8
rectHaut.nbCol = 2
rectHaut.motif = 'I'

rectGros = DessinRectangle()
rectGros.nbLig = 10
rectGros.nbCol = 15
rectGros.motif = 'O'
```

Dans la suite, on utilise le point `.` pour accéder aux attributs d'une variable de type *DessinRectangle*. Les attributs fonctionnent comme des variables normales : on peut lire leur valeur ou la modifier.

Voici un exemple d'utilisation :

```
def dessinerRectangle(rect):
   for iLig in range(rect.nbLig):
      for iCol in range(rect.nbCol):
         print(rect.motif, end = "")
      print()

dessinerRectangle(rectLong)
dessinerRectangle(rectHaut)
dessinerRectangle(rectGros)
```

```
>>>>>>>>>>>>
>>>>>>>>>>>>
>>>>>>>>>>>>
II
II
II
II
II
II
II
II
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
OOOOOOOOOOOOOO
```

Affectations entre enregistrements

Lorsque l'on écrit le nom de la variable qui contient la structure, par exemple pour l'affecter à une autre :

```
rectX = DessinRectangle()
rectX.nbLig = 5
rectX.nbCol = 4
rectX.motif = 'X'
copie = rectX
```

cela correspond à une référence vers l'instance qui s'y trouve. Ainsi, après l'exécution du code ci-dessus, les deux variables pointent vers la même instance de *DessinRectangle*, et on peut donc utiliser indifféremment *rectX* ou *copie* pour manipuler la valeur.

C'est l'appel `DessinRectangle()` qui crée une nouvelle instance à mettre dans des boîtes.

Méthodes

Certains langages permettent d'écrire des fonctions à l'intérieur des types structurés ; on appelle ces fonctions des *méthodes*. Une méthode est équivalente à une fonction, sauf qu'au lieu de faire `fonction(instance, ...)`, elle permet d'écrire `instance.fonction(...)`, donc de faire ressortir l'élément dans l'écriture. Dans le corps de la méthode, l'instance est accessible par un mot-clé particulier.

Voici par exemple la structure précédente, avec une méthode pour le dessin :

```
class DessinRectangle:
   def dessinerRectangle(self):
      for iLig in range(self.nbLig):
         for iCol in range(self.nbCol):
            print(self.motif, end = "")
         print()
```

Avec cette définition, on peut appeler la méthode en écrivant `rectangle.dessinerRectangle()`. On est obligé d'indiquer que `self` est le premier paramètre de la méthode ; car il s'agit de l'instance appelant la fonction.