Fonctions
Cours

Contenus	Capacités attendues
Constructions élémentaires	Être capable de : ▪ Définir une nouvelle fonction, avec ou sans paramètres ▪ Appeler une fonction, avec ou sans paramètres

C'est quoi une fonction en programmation ?

Une fonction en programmation est un bloc de code autonome qui effectue une tâche spécifique.

Voici ses principales caractéristiques :

1. **Nom** : Chaque fonction a un nom qui permet de l'identifier et de l'appeler dans le programme.
2. **Paramètres** : Les fonctions peuvent prendre des arguments (ou paramètres) en entrée, permettant de leur passer des données pour qu'elles les traitent.
3. **Corps** : C'est le code qui définit ce que fait la fonction. Ce code est exécuté chaque fois que la fonction est appelée.
4. **Valeur de retour** : Une fonction peut renvoyer une valeur (ou plusieurs) en utilisant l'instruction return. Cela permet de récupérer le résultat de l'opération effectuée par la fonction.
5. **Portée** : Les variables définies à l'intérieur d'une fonction ont généralement une portée locale, ce qui signifie qu'elles ne sont pas accessibles en dehors de cette fonction.

En résumé, une fonction est un moyen de structurer le code, de le rendre réutilisable, et d'encapsuler des comportements spécifiques.

1. Définir une fonction

Une fonction associe une séquence d'instruction à un nom. Voici un exemple simple d'une fonction :

```
def fonction_calcul() :
    return 3 * 5 + 4
```

La définition commence par le mot-clé **def** suivie par **le nom de la fonction**, puis une **paire de parenthèses** et **deux points**.

Ici nous définissons la fonction mathématique qui au nombre x associent le nombre 3 * 5 + 4.
En Python, le mot-clé **return** permet d'indiquer le résultat de la fonction.
Il est suivi d'une expression et indique que la valeur de cette expression est ***renvoyée*** comme résultat de la fonction.

Le corps de la fonction est écrit en retrait comme on le ferait pour un bloc à l'intérieur d'une instruction **if** ou **for**. Pour exécuter les instructions de la fonction, on appelle la fonction avec la syntaxe suivante :

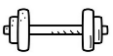
```
1
2 def fonction_calcul() :
3     return 3 * 5 + 4
4
5 resultat = fonction_calcul()
6
7 print(resultat)
8
9
```

Définition de la fonction

Appel de la fonction

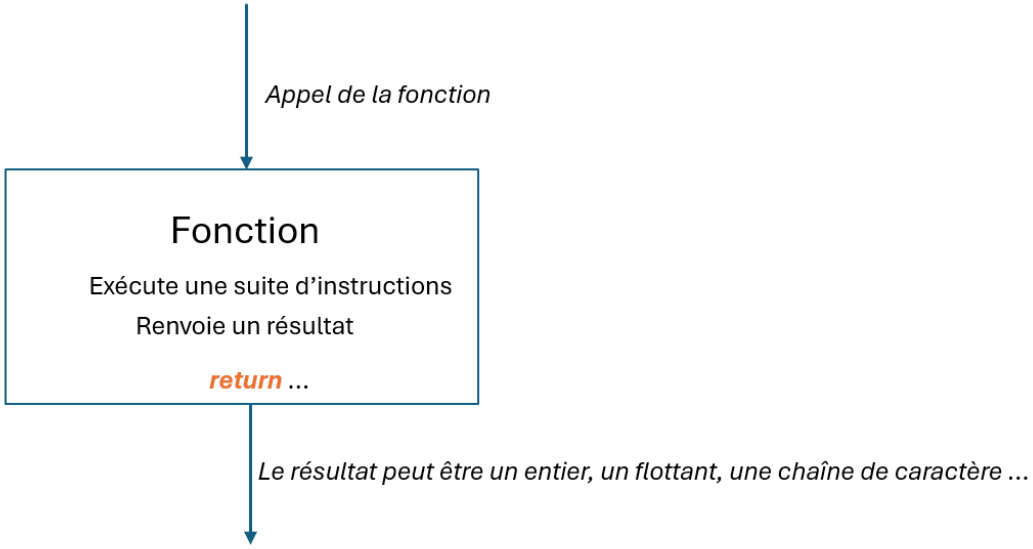
```
Console
>>> %Run -c $EDITOR_CONTENT
 19
>>>
```

Affichage du résultat de la fonction



Activité : Exécuter le code ci-dessus avec Thonny.

Une fonction peut donc être vu comme une boite, exécutant du code.
Elle peut être appelée à tout moment.
Nous pouvons donc **factoriser du code** afin d'éviter de répéter à plusieurs endroits les mêmes instructions (mêmes buts).



Erreurs

Comme dans les chapitres précédents avec for et if c'est l'indentation qui définit le bloc d'instruction formant le corps d'une fonction. Un retrait inattendu comme ci-dessous :

```
1  def truc():
2      valeur_1 = 2
3      valeur_2 = 5
4      total = 0
5      total = valeur_1 + (valeur_1 * valeur_2)
6  print("Total:", total)
7
```

Console

```
>>> %Run -c $EDITOR_CONTENT
 Traceback (most recent call last):
   File "<string>", line 6, in <module>
 NameError: name 'total' is not defined

>>>
```

...produit une erreur immédiate, tandis qu'un retour au niveau de base marque la fin de la définition de la fonction.

Dans le cas précédant, il était plutôt attendu le code suivant :

```
1  def truc():
2      valeur_1 = 2
3      valeur_2 = 5
4      total = 0
5      total = valeur_1 + (valeur_1 * valeur_2)
6      print("Total:", total)
7
8  print("Démarrage du programme...")
9  truc() # Appel de la fonction truc()
10
```

Console

```
>>> %Run -c $EDITOR_CONTENT
 Démarrage du programme...
 Total: 12

>>>
```

2. Fonction avec paramètre

Dans notre fonction précédente, les valeurs valeur_1 et valeur_2 sont fixées respectivement à 2 et 5. Pour effectuer le calcul avec des valeurs quelconques on peut ajouter des paramètres à notre fonction :

```
1  def truc(param_1, param_2):
2      valeur_1 = param_1
3      valeur_2 = param_2
4      total = 0
5      total = valeur_1 + (valeur_1 * valeur_2)
6      print("Total:", total)
7
8  print("Démarrage du programme...")
9  truc(2, 5) # Appel de la fonction truc() avec les paramètres 2 et 5
```

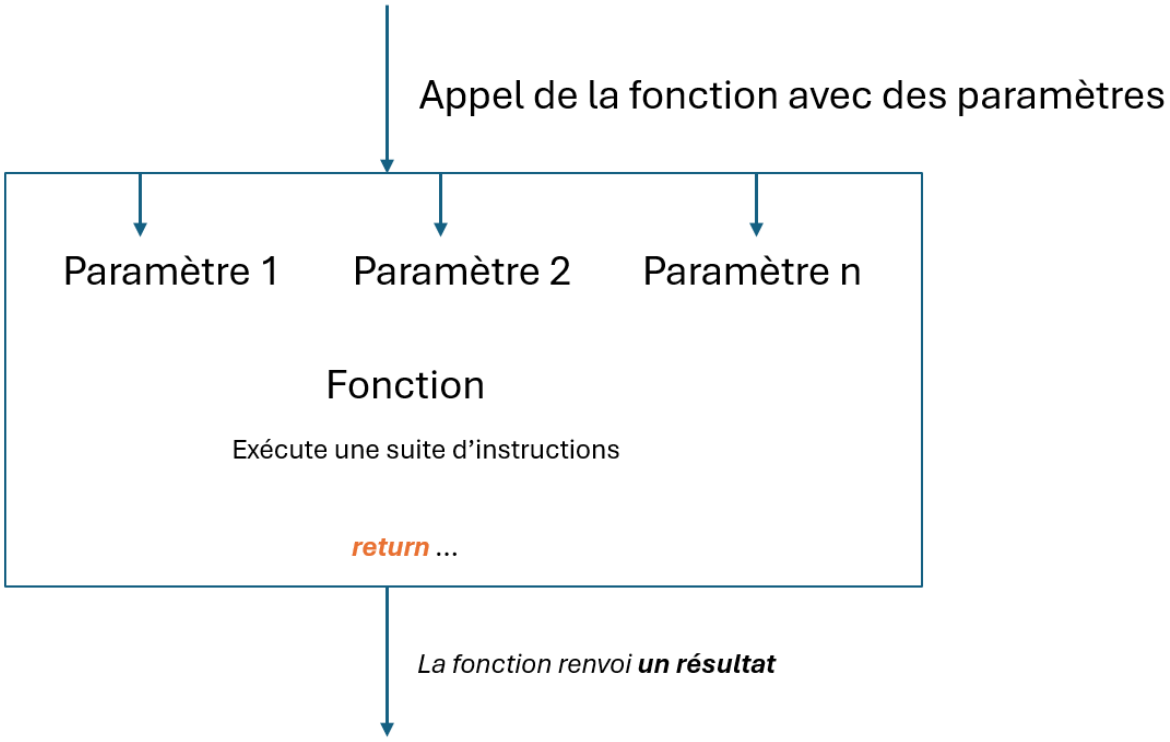
Les paramètres sont désignés par : param_1 et param_2, ajoutés entre les parenthèses. Ils sont ensuite utilisés dans le corps de la fonction à la manière d'une variable. Chaque paramètre désigne une valeur qui n'est pas connue à l'avance et qui pourra être différente à chaque appel de la fonction.

Une version totalement équivalente pourrait être :

```
1  def truc(param_1, param_2):
2      total = 0
3      total = param_1 + (param_1 * param_2)
4      print("Total:", total)
5
6  print("Démarrage du programme...")
7  truc(2, 5) # Appel de la fonction truc() avec les paramètres 2 et 5
```

Console

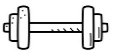
```
>>> %Run -c $EDITOR_CONTENT
 Démarrage du programme...
 Total: 12
>>>
```


Appel de la fonction avec des paramètres
Paramètre 1
Paramètre 2
Paramètre n
Fonction
Exécute une suite d'instructions
return ...
La fonction renvoi un résultat

3. Une fonction sans le mot clé return

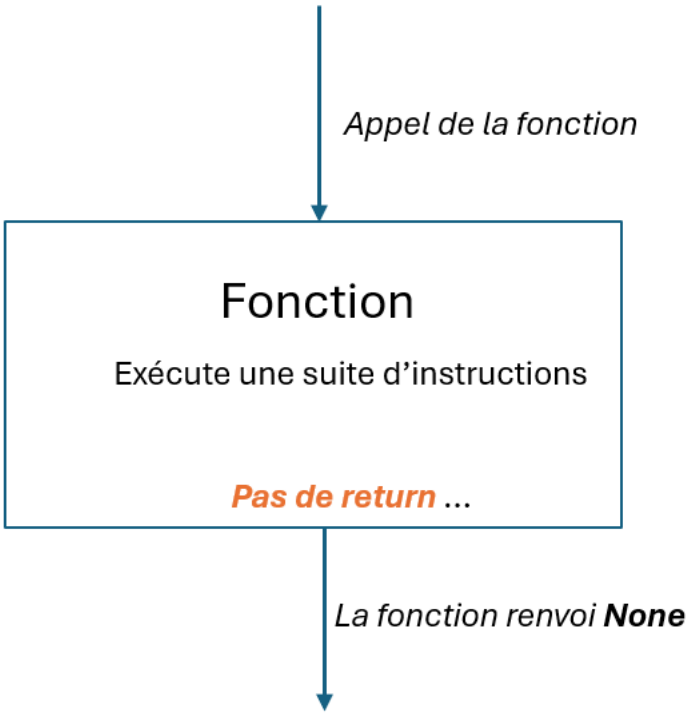
Il est tout à fait possible de définir une fonction sans le mot clé return.

```
1
2  def fonction_sans_return() :
3      x = 2
4      y = 22
5
6
7  resultat = fonction_sans_return()
8
9  print(resultat)
```

 **Activité :** Exécuter le code ci-dessus avec Thonny. Que constatez-vous ?

None est défini comme la représentation de l'absence de valeur.
None est utilisé pour signifier l'achèvement d'une fonction **sans retour**.

Vous remarquerez que même si nous n'utilisons pas l'instruction **return**, la fonction retourne par défaut la valeur **None**.



4. Fonction et procédure

Une fonction qui ne renvoie pas de résultats est aussi appelée une procédure. Même s'il n'y a pas de différence entre les deux notions en Python, elles répondent à des objectifs très différents.
Une procédure réalise une action, par exemple, un affichage, une écriture, dans un fichier ou encore un dessin. Un appel de procédure constitue une instruction d'un programme et peut donc se trouver à tout endroit où on s'attend à trouver des instructions.

Une fonction avec résultat a pour objectif de calculer et mettre à disposition une valeur. Un appel d'une telle fonction constitue une expression et peut donc être utilisée soit à un endroit où on s'attend à trouver une expression, par exemple à droite du symbole d'affectation, soit comme partie d'une expression plus grande :

```
def addition(x, y) :
    return x + y

valeur = 12 + addition(3, 2) *2
```

Pour réunifier ces deux concepts, une fonction Python renvoie en réalité toujours une valeur. En l'absence de return, une valeur spéciale, notée None, est renvoyée implicitement lorsque la fin de la fonction est atteinte.

5. Renvoyer plusieurs résultats

Si une fonction doit renvoyer plusieurs résultats, elle peut le faire en utilisant un n-uplet. Un n-uplet regroupe plusieurs valeurs entre parenthèses et séparées par des virgules.

```
>>> (1+2, 3*4)
(3, 12)
```

On peut écrire en particulier une fonction qui renvoie un n-uplet, ce qui est une façon de renvoyer plusieurs valeurs.

```
def foo(x) :
    return (x * 10, x // 10)
```

Pour récupérer les différentes composantes d'un n-uplet, on peut l'affecter à un n-uplet de variables. Ainsi, on peut écrire :

```
>>> (a, b) = foo(42)
```

Ce qui a pour effet de donner la valeur 420 à la variable a et la valeur 4 à la variable b.

6. Variable locale à une fonction

Comme tout autre morceau de code Python, le corps d'une fonction peut introduire des variables pour ses calculs intermédiaires. Supposons par exemple que l'on veut définir une fonction qui calcule le produit de n premier entier (appeler la factorielle de n). On peut le faire avec une variable locale à la fonction et une boucle.

```
def fact(n) :
  f = 1
  for i in range( 2, n+1) :
    f = f * i
  return f
```

$1! = 1$

$2! = 1 \times 2 = 2$

$3! = 1 \times 2 \times 3 = 6$

$4! = 1 \times 2 \times 3 \times 4 = 24$

$10! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10 = 3\,628\,800$

Cette variable locale f est définie à l'intérieur de la fonction, au moment de son initialisation, c'est-à-dire l'instruction f = 1, et elle disparaît à la fin de l'exécution de la fonction. On peut observer que la variable locale f n'existe pas à l'extérieur de la fonction fact, même après un appel à fact.

```
>>> fact(4)
24
```

```
>>> f
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'f' is not defined
```

Même si une variable f est déjà définie avant l'appel à fact(4), elle conservera sa valeur.

```
>>> f = 42
>>> fact(4)
24
>>>f
42
```

Pendant l'exécution de fact(4), il existe simultanément deux variables qui s'appellent f : la variable définie à l'extérieur de la fonction et la variable locale à la fonction. Pendant l'exécution de la fonction, seule la variable locale est accessible. On dit qu'elle *masque* la première variable.

Lorsqu'une valeur est passée en argument à une fonction, elle est reçue comme une variable locale à la fonction, dont la durée de vie correspond exactement l'exécution de la fonction. Comme toute autre variable locale, elle peut être modifiée par la fonction.

```
def incremente(x) :
    x = x + 1
```

Il est important de comprendre que cette variable qui reçoit la valeur de l'argument est une nouvelle variable, sans rapport avec les variables qui existent au moment de l'appel, y compris lorsque l'appel se fait sur la valeur d'une variable.

```
>>>x = 1
>>>incremente(x)
>>>x
1
```

7. Sortie anticipée

L'instruction **return** peut apparaître n'importe où dans le code d'une fonction. Son effet est non seulement de renvoyer le résultat de la fonction mais également d'en interrompre l'exécution : aucune des instructions éventuellement présentes dans la suite du code de la fonction n'est alors plus exécutée. Cela sert notamment à interrompre l'exécution d'une fonction dans un cas où le résultat serait déjà connu.

```
def oa(x) :
    if f1(x) : return True
    if not f2(x) : return False
    return f3(x)
```