

7 – Fonctions

Introduction



ACTIVITÉ 1

On souhaite réaliser une application. Lors de l'exécution de notre application, avant d'arriver à l'écran de menus, il est habituel d'afficher un écran d'accueil appelé *splash screen* affichant le nom de l'application, une référence à l'équipe de production.

L'application affiche un écran d'accueil le nom de l'application, sa version et le nom de l'équipe de production. Après le chargement des données, on indiquera à l'utilisateur qu'il doit appuyer sur Entrée pour passer à la suite. Lorsque l'utilisateur valide, afficher un menu (par exemple "Nouvelle Partie", "Sauvegardes", "Configuration").

CORRECTION

Le programme réalisé est peu satisfaisant. Il comporte clairement trois grandes parties différentes mais tout est écrit en un seul bloc. Il serait pratique de structurer le code en trois sous fonctionnalités :

- `afficher_splash`,
- `charger_data` et
- `afficher_menu`.

Nous allons voir dans cette leçon comment rendre le code plus élégant en définissant des **fonctions**, c'est à dire des fragments de codes réutilisables réalisant une tâche donnée pouvant dépendre d'un certain nombre de paramètres.

Définir une fonction

Une fonction associe une séquence d'instructions à un nom.

Exemple

Suite de l'activité 1. Voici par exemple un exemple simple d'une fonction `afficher_menu` qui affiche un menu composé de trois lignes : "Nouvelle Partie", "Sauvegardes", "Configuration".

```
def afficher_menu():  
    print("=====  
    print("Nouvelle Partie")  
    print("-----")  
    print("  Sauvegardes")  
    print("-----")  
    print(" Configuration")  
    print("=====
```

La **définition** commence par

- le mot-clé `def`, suivi par
- le nom de la fonction (ici `afficher_menu`), puis
- une paire de parenthèses et
- deux points.

Exemple

Suite de l'activité 1. La fonction est constituée d'un bloc de 7 instructions : le *corps de la fonction*.

Comme pour un bloc `for` ou `if`, le corps est écrit en retrait.

Pour exécuter la fonction `afficher_menu`, on **appelle** la fonction avec la syntaxe suivante :

Exemple

Suite de l'activité 1. Voici comment appeler la fonction `afficher_menu` pour que le menu s'affiche :

```
afficher_menu()
```

Appeler la fonction produit le même résultat que si on avait exécuté le corps de la fonction directement. On peut visualiser son exécution sur [Python Tutor](#).



Fonction avec paramètres

Les fonctions peuvent dépendre d'un **paramètre**. C'est-à-dire d'une valeur qui sera précisée à **chaque** appel de la fonction.

Cette valeur peut être différente d'un appel à un autre.

Exemple

Suite de l'activité 1. Soit la fonction `charger_data` qui affiche une barre de chargement. Imaginons que la durée du chargement doive **varier** d'un appel à un autre.

Pour réaliser cela, on définit un paramètre qu'on nomme `duree`.

Chaque appel se fait avec une valeur concrète dans la parenthèse. Par exemple `charger_data(2)` pour signifier une durée de 2 secondes.

De son côté, la fonction crée une variable locale `duree` qui est initialisée avec la valeur concrète passée en paramètre. Par exemple après l'appel `charger_data(2)`, on a l'affectation suivante $duree \leftarrow 2$. On obtient donc en mémoire locale à la fonction :

Mémoire locale à la fonction `charger_data`

	...
duree	2
	...

Puis lors de l'exécution de la fonction, chaque référence à la variable `duree` fait pointer vers la variable concrète locale à la fonction.

```
[2]: def charger_data(duree):
      print("chargement en", duree, "secondes")
```

```

      fraction_duree = duree / 16
      from time import sleep
      print(f"{' '*15}", end="")
      for i in range(16):
          print(f"\r[{'#'*i}]", end="")
          sleep(fraction_duree)
      print("\nfin\n")
```

```

charger_data(1) # la fonction s'exécute avec `duree` qui vaut 1
charger_data(2) # la fonction s'exécute avec `duree` qui vaut 2
charger_data(4) # la fonction s'exécute avec `duree` qui vaut 4
```

```

chargement en 1 secondes
[#####]
fin
```

```

chargement en 2 secondes
[#####]
fin
```

```

chargement en 4 secondes
[#####]
fin
```

Fonction à plusieurs paramètres

Il peut être nécessaire d'avoir des fonctions avec **plusieurs** arguments. Pour ajouter des arguments, on ajoute des noms de variables dans les parenthèses de la fonction.

Exemple

Pythagore. Par exemple pour savoir si un triangle est rectangle, il faut connaître les longueurs de ses **trois côtés**. On peut donc imaginer une fonction `test_pythagore()` qui indique si oui ou non un triangle est rectangle.

Pour s'adapter à chaque triangle, il est nécessaire d'indiquer à chaque appel de la fonction les valeurs des trois longueurs. La fonction nécessite donc **3 arguments** : un pour chaque longueur.

```
def test_pythagore(longueur_1, longueur_2, longueur_3):  
    ...
```

Exemple

Fonctions prédéfinies. Depuis le début de l'année, vous avez utilisé et manipulé de nombreuses fonction qui sont prédéfinie en Python.

Par exemple :

- `print()` pour afficher du texte (qui admet en argument un n-uplet (tuple))
- `int()` pour convertir en nombre entier (qui admet en argument la chaîne de caractère et (facultatif) la base)
- `float()` pour convertir en nombre décimal une chaîne de caractère

- `input()` pour afficher un texte (argument de la fonction) et renvoyer le texte écrit par l'utilisateur.

REMARQUE

`break` ou `continue` ne sont pas des fonctions, mais des *mots clés*.
Il n'y a pas de parenthèses après les mots-clés !

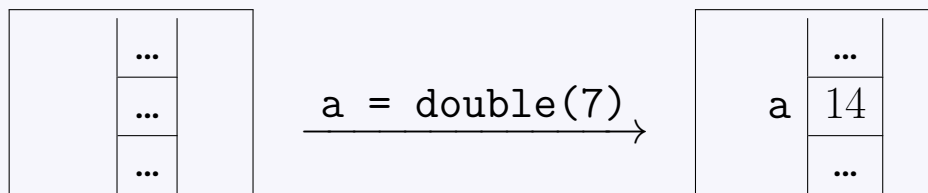
Renvoyer un résultat

Lorsqu'une fonction termine son exécution, tout son espace mémoire local est supprimé. Ainsi, **tous les résultats** obtenus par la fonction sont effacés.

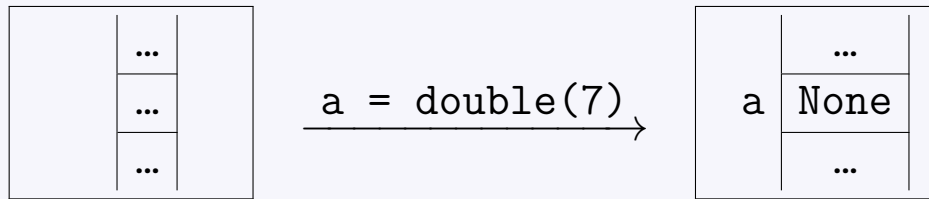
Mais avant cette suppression, la fonction peut **renvoyer** un résultat. C'est-à-dire que l'appel de la fonction *devient* le résultat renvoyé.

Exemple

Double d'un nombre. Imaginons la fonction `double(x)` qui admette un paramètre `x` et calcule son double. On souhaite donc qu'après l'instruction suivante `a = double(7)` la variable `a` contienne la valeur 14.



Mais, si la fonction ne renvoie rien, alors on aura plutôt :



La variable `a` existe mais ne contient... rien du tout.

C'est pour cela que la fonction **doit renvoyer un résultat**!

Pour renvoyer un résultats, on utilise le mot clé `return` suivi du résultat.

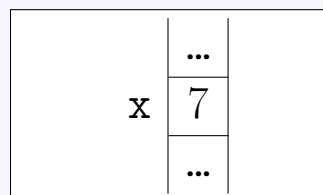
Exemple

Double d'un nombre (suite). La fonction `double` peut être implémentée comme ceci :

```
def double(x):
    reponse = x * 2
    return reponse
```

```
a = double(7)
print(a)
```

1. L'instruction `a = double(7)` appelle la fonction avec le paramètre 7.
2. Un espace mémoire local à la fonction est réservé et la variable `x` est initialisée avec la valeur 7



3. La première ligne de la fonction initialise la variable `reponse` avec la valeur égale au résultat de `x * 2`. Comme `x` contient le nombre

7, $x * 2$ est évalué à 14 et donc on a l'affectation $\text{reponse} \leftarrow 14$.

	...
x	7
reponse	14
	...

4. En écrivant `return reponse`, la fonction se termine **immédiatement** en renvoyant la valeur contenue dans la variable `reponse` : 14.
5. Le résultat du calcul de `double(7)` devient donc égal à 14. Ainsi, l'affectation $a \leftarrow \text{double}(7)$ devient $a \leftarrow 14$.

	...
a	14
	...

Exemple

Exécution en ligne.

Voici un lien pour visualiser
l'instruction `print(double(10)`
* 3) : [Python Tutor](#)



REMARQUE

Une fonction qui ne renvoie rien s'appelle une *procédure*.

En l'absence de `return`, la procédure renvoie malgré tout `None`.

Renvoyer plusieurs résultats

Une fonction peut aussi renvoyer plusieurs résultats. Pour cela, elle revoie un seul objet : un *n-uplet*.

REMARQUE

Point de vocabulaire à propos des *n-uplets*.

Nombre d'éléments	Exemple	Nom usuel
2	(4, 5)	couple
3	(3, 1, 4)	triplet
5	(2, 4, 6, 8, 10)	quintuplet
7	(0, 1, 2, 3, 4, 5, 6)	septuplet

Pour renvoyer un *n-uplet*, il suffit d'indiquer après le mot-clé `return` le *n-uplet*. En Python, un *n-uplet* est un objet de type `tuple` qui s'obtient grâce à des **parenthèses**.

Exemple

Ainsi on peut créer une fonction `calcul` qui renvoie le double et le triple d'un nombre entier `n`

```
def calcul(n):  
    double = n * 2  
    triple = n * 3
```

```
return (double, triple)
```

REMARQUE

Un décorateur Python tolère la syntaxe *sans les parenthèses* : `return double, triple`.

Ce décorateur fonctionne aussi pour l'affectation. Les deux syntaxes suivantes sont donc équivalentes :

```
(d_2, t_2) = calcul(2)    # tuple explicite
d_10, t_10 = calcul(10)  # décorateur: tuple implicite
```

Ce code modifie l'état des variables `d_2`, `t_2`, `d_10` et `t_10`. Il effectue les affectations suivantes :

...
...
...

```
(d_2, t_2) = calcul(2)
d_10, t_10 = calcul(10)
```

	...
d_2	4
t_2	6
d_10	20
t_10	30
	...

Variables locales à une fonction

Une variable créée en dehors d'une fonction n'a pas d'existence au sein de la fonction. Et réciproquement, une variable créée dans une fonction, n'a pas d'existence en dehors de la fonction.

C'est ce qu'on appelle variable *globale* et variable *locale*.



ACTIVITÉ 2

Soit le code ci-dessous :

```
def modifier():
    a = 100
```

```
a = 1
modifier()
print(a)
```

Quelle valeur va être affichée à l'écran ? Proposer une explication.

CORRECTION

La variable `a` initialisée à 1 avant l'appel de la fonction `modifier` vaut **toujours** 1 après l'appel.

En effet, c'est la variable `a` *locale* (à l'intérieur de la fonction) qui a été modifiée et non pas la variable *globale* (celle à l'extérieur).

Tout ce passe comme si ces deux variables sont différentes (ce qui est le cas en mémoire d'ailleurs!)

Avant l'appel

	...	
a	1	
	...	

Pendant l'appel

	...		modifier()
a	1	a	100

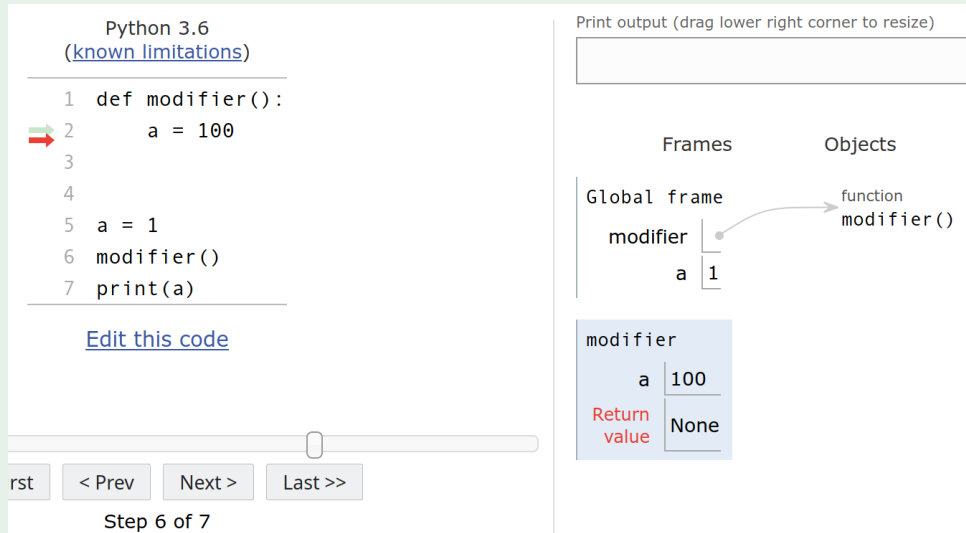
Après l'appel

	...	
a	1	
	...	

Comme on le voit, il y a bien **deux** variables `a` situées à des endroits différents. Chaque zone est imperméable à l'autre. Seule les *paramètres* des fonctions et les valeurs *renvoyées* permettent une communication.

REMARQUE

L'application en ligne Python Tutor permet de voir l'état de la mémoire lors de l'exécution d'un code.



The screenshot shows the Python Tutor interface for Python 3.6. The code being executed is:

```

1 def modifier():
2     a = 100
3
4
5 a = 1
6 modifier()
7 print(a)

```

The execution is at line 2, where `a = 100` is being assigned. The memory state is shown on the right:

- Global frame:** Contains `modifier` pointing to the `function modifier()` object and `a` with value `1`.
- function modifier():** Contains `a` with value `100` and a `Return value` of `None`.

Navigation buttons at the bottom include 'rst', '< Prev', 'Next >', and 'Last >>'. The progress indicator shows 'Step 6 of 7'.



ACTIVITÉ 3

Qu'affiche l'exécution du code suivant ?

```

def incrementer(x):
    x = x + 1
    print("dans la fonction, x:", x)

x = 1
print("au moment de l'appel, x:", x)
incrementer(x)
print("après l'appel, x:", x)

```

CORRECTION

1. Au moment de l'appel à la fonction `incrementer`, la variable

- `x` vaut 1. Un premier affichage indique donc au moment de l'appel, `x: 1`.
2. La appel à la fonction `incrémenter` avec pour argument `x` a pour effet immédiat de créer une variable **locale** `x` qui est initialisée avec la valeur 1 contenue dans la variable **globale** `x`. Puis la première instruction de la fonction modifie cette variable *locale* qui vaut maintenant 2. Enfin, la deuxième instruction affiche la valeur de de cette variable *locale* et affiche donc dans la fonction, `x: 2`.
 3. La fonction se termine et la variable *locale* `x` disparaît. En revanche, la variable globale `x` est toujours présente et n'a absolument pas été modifiée. Et la dernière instruction affiche après l'appel, `x: 1`.

Sortie anticipée

Dès que le mot clé `return` est rencontré, l'exécution de la fonction s'arrête **immédiatement**.

La fonction renvoie alors le ou les résultats situés à droite du mot-clé.



ACTIVITÉ 4

Qu'affiche le programme ci-dessous ?

```
def sortie(n):  
    if n > 100:  
        return True  
    return False  
  
print( (sortie(101), sortie(99)) )
```

CORRECTION

Le programme affiche un n-uplet composé de (1) le résultat de l'appel à `sortie(101)` et de (2) le résultat de l'appel à `sortie(99)`.

Commençons par le (2). La fonction s'initialise avec `n` qui vaut 99. Comme 99 est inférieur à 100, on a la condition `n > 100` qui est fausse. Le bloc conditionnel n'est pas exécuté. Le programme *saut* à la ligne 3 qui renvoie `False`.

Pour le (1), la variable `n` est initialisée à 101. La condition `n > 100` est vraie. Le bloc conditionnel est exécuté. La ligne 2 est un renvoi qui **interrompt immédiatement** le programme. La fonction renvoie la valeur `True`.

Pour conclure, le programme affiche le tuple : `(True, False)`.

Exercices**ACTIVITÉ 5**

Définir une fonction `test_Pythagore` qui prend trois entiers a , b et c en arguments et renvoie un booléen indiquant si $a^2 + b^2 = c^2$, ou $b^2 + c^2 = a^2$ ou $c^2 + a^2 = b^2$.

**ACTIVITÉ 6**

Définir une fonction `valeur_absolue` qui prend un entier en argument et renvoie sa valeur absolue.

**ACTIVITÉ 7**

Créer une fonction `multiples` pour qu'elle prenne la limite en argument plutôt que d'utiliser la valeur 999. En déduire une fonction `multiples_3_ou_5(borne_sup)` qui renvoie la somme des multiples de 3 ou 5 inférieurs ou égaux à `borne_sup`.

**ACTIVITÉ 8**

Écrire une fonction `max2(a, b)` qui renvoie le plus grand des deux entiers `a` et `b`.

**ACTIVITÉ 9**

En supposant la fonction `max2` précédente disponible, **écrire** une fonction `max3(a, b, c)` qui utilise la fonction `max2` de l'exercice précédent et qui renvoie le plus grand des trois entiers `a`, `b`, `c`.

**ACTIVITÉ 10**

Écrire une fonction `puissance(x, k)` qui renvoie `x` à la puissance `k` (utilisation de l'opérateur `**` interdit! évidemment...). On utilisera une boucle `for` pour faire le calcul.

On suppose que $k \geq 0$ et on rappelle que $x^0 = 1$.

**ACTIVITÉ 11**

Écrire une fonction `est_bissextile(annee)` qui renvoie un booléen indiquant si l'année `annee` est une année bissextile.

On rappelle qu'une année bissextile est une année multiple de 4 mais pas de 100, ou multiple de 400.

**ACTIVITÉ 12**

Écrire une fonction `nb_jours_annee(annee)` qui renvoie le nombre de jour de l'année `annee`. La fonction `est_bissextile` de l'exercice précédent est disponible et vous pouvez l'utiliser (ce qui est bien pratique quand même).

**ACTIVITÉ 13**

En supposant que le mois `mois` est un entier compris entre 1 (pour janvier) et 12 (décembre), **écrire** une fonction `nb_jours_mois(annee, mois)` qui renvoie le nombre de jours dans le mois `mois` de l'année `annee`. La fonction `est_bissextile` de l'exercice précédent est disponible (ce qui est bien pratique).

**ACTIVITÉ 14**

En supposant les fonctions `nb_jours_annee` et `nb_jours_mois` disponibles, **écrire** une fonction `nb_jours(j_debut, m_debut, a_debut, j_fin, m_fin, a_fin)` qui renvoie le nombre de jours compris entre deux dates données (incluses).