PYTHON – Fonctions

Introduction

Très souvent, un programmeur crée ses **propres** fonctions.

En effet, une application, surtout si elle est longue, a toutes les chances de devoir **procéder aux mêmes traitements à plusieurs endroits** de son déroulement.

Par exemple, la saisie d'une réponse par oui ou par non peut être répétée dix fois à des moments différents de la même application, pour dix questions différentes.

La manière la plus évidente de programmer ce genre de choses, c'est de **répéter le code** correspondant **autant de fois que nécessaire**.

- Même si la structure d'un programme écrit de cette manière peut paraître simple, elle est en réalité inutilement lourde. Elle contient des répétitions, et le programme peut devenir parfaitement **illisible**.
- En plus, une telle structure pose des **problèmes de maintenance** : car en cas de modification du code, il va falloir traquer toutes les apparitions plus ou moins identiques de ce code pour faire convenablement la modification !

Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à **regrouper les instructions qui le composent dans un module séparé**.

Il ne restera alors plus qu'à appeler ce groupe d'instructions (qui n'existe donc désormais qu'en un exemplaire unique) à chaque fois qu'on en a besoin. Ainsi, la lisibilité est assurée ; le programme devient modulaire, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l'application.

Le corps du programme s'appelle alors la **procédure principale**

Les groupes d'instructions auxquels on a recours s'appellent des **fonctions.**

Syntaxe

Prenons un exemple de question à laquelle l'utilisateur doit répondre par oui ou par non.

```
print("êtes vous marié ?")
Rep1 = ""
while Rep1 != "Oui" and Rep1 != "Non":
   Rep1 = input("Entrez Oui ou Non")
...
print("Avez-vous des enfants ?")
Rep2 = ""
while Rep2 != "Oui" and Rep2 != "Non":
   Rep2 = input("Entrez Oui ou Non")
```

La seule chose qui change, c'est l'intitulé de la question, et le nom de la variable dans laquelle on range la réponse (Rep1, Rep2)

La solution va consister à isoler les instructions demandant une réponse par Oui ou Non.

Ainsi, on évite les répétitions inutiles, et on découpe notre problème en petits morceaux autonomes.

Nous allons donc créer notre propre fonction, que nous appellerons RepOuiNon, et dont le rôle sera de renvoyer la réponse (oui ou non) de l'utilisateur.

```
def RepOuiNon():
        Reponse =  ""
        while Reponse!= "Oui" and Reponse!= "Non"  :
                Reponse = input("Entrez Oui ou Non")
        return Reponse
```

- On utilise le mot-clé **def** pour écrire une fonction
- Les **parenthèses** RepOuiNon () sont obligatoires.
- N'oubliez pas les **deux points**
- Le nouveau mot-clé **return** indique la valeur renvoyée par la fonction (ici, la valeur de la variable **`Reponse`**).

Une **fonction s'écrit toujours en-dehors de la procédure principale**. Ce qu'il faut comprendre, c'est que les lignes de codes contenues dans la fonction sont en quelque sorte des satellites, qui existent en dehors du traitement lui-même. Simplement, elles sont à sa disposition, et il pourra y faire appel chaque fois que nécessaire.

Si l'on reprend notre exemple, une fois notre fonction RepOuiNon écrite, le **programme principal** comprendra les lignes :

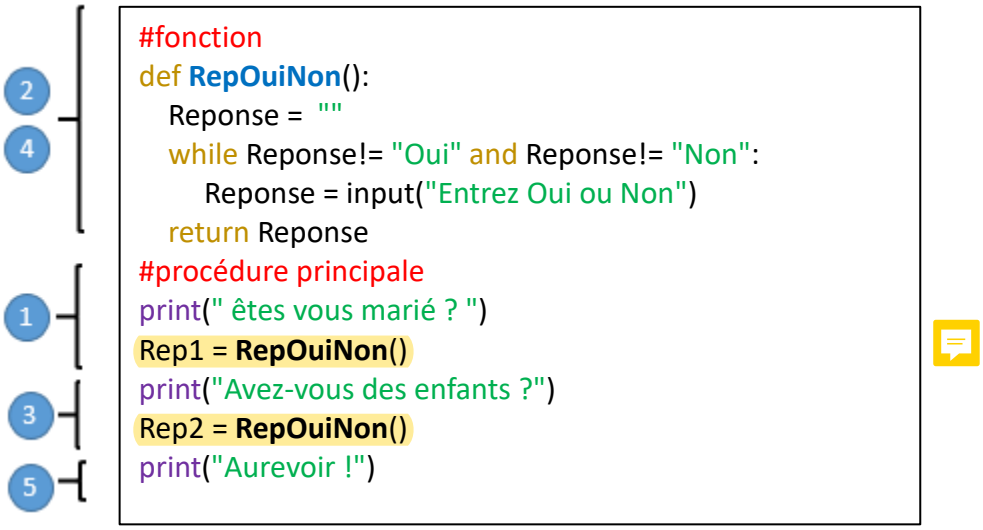
```
print("  êtes vous marié ? ")
Rep1 = RepOuiNon()
print(" Avez-vous des enfants ?")
Rep2 = RepOuiNon()
```

On a ainsi évité les répétitions inutiles, et s'il y avait un bug, il suffirait de faire une seule correction dans la fonction pour que ce bug soit éliminé de toute l'application.

Ordre de lecture du programme

Attention, dans un programme, les fonctions sont toujours écrites en premier, mais elles ne sont pas lues en premier.

Les instructions qui sont lues en premier sont celles qui sont dans la procédure principale. La fonction est lue à partir du moment où elle est appelée par une instruction de la procédure principale



NB : On a **capturé la valeur renvoyée par le fonction** (Reponse), en **plaçant une variable devant l'appel de la fonction (**Rep1 et Rep2)

Résultat dans la console :

```
 êtes vous marié ?
Entrez Oui ou Non : Oui
 Avez-vous des enfants ?
Entrez Oui ou Non : Non
>>>
```

Passage d'arguments

la fonction RepOuiNon peut encore être améliorée : puisque avant chaque question, on doit écrire un message (print(" êtes vous marié ? ") par exemple), autant que cette écriture du message figure directement dans la fonction appelée.

Cela implique deux choses :

- lorsqu'on appelle la fonction, on doit lui préciser quel message elle doit afficher avant de lire la réponse
- la fonction doit être « prévenue » qu'elle recevra un message, et être capable de le récupérer pour l'afficher.

En programmation, on dira que **le message devient un argument (ou un paramètre) de la fonction**. Quitte à construire nos propres fonctions, nous pouvons donc construire nos propres arguments ! on peut d'ailleurs passer autant d'arguments qu'on veut, et créer des fonctions avec deux, trois, quatre, arguments…

La fonction sera dorénavant écrite comme suit :

```
#fonction
def RepOuiNon(Msg):
   print(Msg)
   Reponse = ""
   while Reponse!= "Oui" and Reponse!= "Non"  :
      Reponse = input("Entrez Oui ou Non")
   return Reponse
```

Il y a donc maintenant entre les parenthèses une variable Msg, qui signale à la fonction qu'un argument doit lui être envoyé à chaque appel. Voici l'appel dans la procédure principale :

```
#procédure principale
Rep1 = RepOuiNon("êtes vous marié ?")
Rep2 = RepOuiNon("Avez-vous des enfants ?")
print("Aurevoir !")
```

Récapitulatif sur la création de fonctions avec return

```
def nom_de_la_fonction(parametre1, parametre2, parametre3, parametreN):
    # instructions diverses
    return valeur
```

On crée une fonction selon le schéma suivant :

- **def**, mot-clé qui est l'abréviation de « define » (définir, en anglais)

  def constitue le début de toute construction de fonction.
- Le **nom de la fonction**, qui se nomme exactement comme une variable.
  N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La **liste des paramètres** qui seront fournis lors d'un appel à la fonction.
  - On peut avoir aucun paramètre, 1 seul paramètre ou plusieurs paramètres
  - Les paramètres sont séparés par des **virgules**
  - la liste est encadrée par des **parenthèses** (les espaces sont optionnels mais améliorent la lisibilité).
    Les parenthèses sont obligatoires, même si votre fonction n'a aucun paramètre
- Les **deux points** qui clôturent la ligne
- Le **return** est toujours la **dernière** instruction et indique la valeur renvoyée par la fonction

  Il est donc inutile de placer des instructions après le mot clé return, elles ne seront jamais lues.

Exemple : Le code pour multiplier deux nombres dans une fonction serait :

```
#fonction qui multiplie un nombre par un autre nombre
def multiplication(nombre1,nombre2):
    result = nombre1*nombre2
    return result
#procédure principale
resultat1=multiplication(2,3)
resultat2=multiplication(3,4)
resultat3=multiplication(8,89)
print(resultat1)
print(resultat2)
print(resultat3)
```

```
6
12
712
>>>
```

NB : On a **capturé la valeur renvoyée par le fonction** (result), en **plaçant une variable devant l'appel de la fonction (**resultat1 et resultat2 et resultat3).

Après exécution de l'instruction multiplication(2,3), la variable resultat1 contiendra, 2 *3, c'est-à-dire 6.

Avantage : on peut appeler plusieurs fois la fonction multiplication, afin de multiplier facilement d'autres nombres

Cas particulier : fonctions sans return, aussi appelé procédure

On distingue deux sous-programmes différents : les fonctions et les procédures.

- Une fonction a pour but de renvoyer un résultat.
- Une procédure ne renvoie aucun résultat (affichage, échange, saisie...).

```
#Programme <nom fichier>.py
# fonction sans return (procédure)
def hi():
   print("Hi !")
#procédure principale
hi()
```

Résultat dans la console :

```
Hi !
>>>
```

```
#Programme <nom fichier>.py
# fonction sans return (procédure)
def afficheMax (a,b):
   if a>b:
      print("La valeur maximale est ", a)
   else:
      print("La valeur maximale est ", b)
#procédure principale
nb1 = input("entrer le premier nombre : ")
nb2 = input("entrer le deuxième nombre : ")
afficheMax (nb1,nb2)
```

Résultat dans la console :

```
entrer le premier nombre : 23
entrer le deuxième nombre : 12
La valeur maximale est  23
>>>
```

Appel de plusieurs fonctions dans un même programme

Un même programme peut contenir plusieurs fonctions différentes :

```
#Programme bonjour.py
# fonctions sans return
def hi():
    print("Hi !")
def hello():
    print("Hello !")
#fonction avec return
def bonjour():
    return "bonjour"
#procédure principale
hi()
hello()
result = bonjour() #appel de la fonction et affectation à la variable result
print("je vous dis", result)
print("je vous dis", bonjour()) #On appelle directement le résultat de la fonction
```

Résultat dans la console :

```
Hi !
Hello !
je vous dis bonjour
je vous dis bonjour
```

Quand créer une fonction dans son code ?

Le plus important, c'est d'acquérir le réflexe de **constituer systématiquement les fonctions adéquates quand on doit traiter un problème donné**, et de repérer la bonne manière de découper son programme en différentes fonctions pour le rendre léger, lisible et performant.

On parle d'ailleurs à ce sujet de **factorisation du code,** : c'est une manière de parler mathématique. Factoriser un calcul, cela veut dire regrouper des éléments communs pour éviter qu'ils ne se répètent.

Valeurs par défaut des paramètres

On peut également préciser une valeur par défaut pour les paramètres de la fonction, cette valeur sera affectée au paramètre si l'utilisateur de votre fonction ne le précise pas.

```
def nom_de_la_fonction(parametre1, parametre2=valeur par défaut):
    # Bloc d'instructions
```

Prenons un exemple de code pour une table de multiplication

Vous pouvez par exemple indiquer que le nombre maximum d'affichages doit être de 10 par défaut

```
def table(nb, max=10):
    i = 1
    while i <= max:,
        print(i , "*", nb, "=", i  * nb)
        i += 1
# bloc principal
table(5,20)
table(5)
```

```
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
```

À présent, vous pouvez **appeler la fonction de deux façons** :

- soit en précisant le numéro de la table et le nombre maximum d'affichages,
- soit en ne précisant que le numéro de la table (table(5)). Ici, max vaudra 10 par défaut

Appeler des paramètres par leur nom

on peut appeler des paramètres par leur nom.

Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut.

Prenons un exemple de définition de fonction

```
def fonc(a=1, b=2, c=3, d=4, e=5):
        print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)
```

vous avez de nombreuses façons d'appeler cette fonction :

Instruction	**Résultat**	
fonc()	a=1 b=2 c=3 d=4 e=5	Quand tous les paramètres ont une valeur par défaut, vous pouvez appeler la fonction sans paramètre
fonc(**4**)	a=**4** b=2 c=3 d=4 e=5	Quand on utilise des paramètres sans les nommer, il faut respecter l'ordre d'appel des paramètres
fonc(b=8, d=5)	a=1 b=8 c=3 d=5 e=5	Si vous voulez changer la valeur d'un paramètre, vous tapez son nom, suivi d'un signe égal puis d'une valeur
fonc(b=35, c=48, a=4, e=9)	a=4 b=35 c=48 d=4 e=9	peu importe l'ordre d'appel des paramètres

docstring

La docstring, que l'on pourrait traduire par une **chaîne d'aide**, est une chaîne de caractères qui permet d'ajouter quelques lignes d'explications à la fonction :

- Elle se place juste en-dessous de la définition de la fonction.
- on indente cette chaîne et on la met entre **triple guillemets**.
- Si la chaîne figure sur une seule ligne, on pourra mettre les trois guillemets fermants sur la même ligne ; sinon, on préférera sauter une ligne avant de fermer cette chaîne, pour des raisons de lisibilité.
- Tout le texte d'aide est indenté au même niveau que le code de la fonction.

```
def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb
    de 1*nb à max*nb
    (max >= 0)"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Si vous tapez **help(table)**, c'est ce message que vous verrez apparaître.

Documenter vos fonctions est également une bonne habitude à prendre

```
help (table)
```

Résultat dans la console :

```
Help on function table in module __main__:
table(nb, max=10)
    Fonction affichant la table de multiplication par nb
    de 1*nb Ó max*nb
    (max >= 0)
```

Variables locales et globales

Lorsque nous définissons des variables à l'intérieur d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même.

On dit que ces variables sont des **variables locales** à la fonction.

Les variables définies à l'extérieur d'une fonction sont des **variables globales**.
Leur contenu est « visible » de l'intérieur d'une fonction, mais **la fonction ne peut pas le modifier**

```
def essai():
        p = 20
        print(p, q) # affiche 20 38
p, q = 15, 38
essai()
print(p, q) # affiche 15 38
```

```
20 38
15 38
```

À l'intérieur de la fonction essai() :

- une variable **p** est définie, avec **20** comme valeur initiale. Cette variable **p** qui est définie à l'intérieur d'une fonction sera donc une variable locale.
- la variable globale **q** est appelée dans le print. son contenu est « visible » de l'intérieur d'une fonction, mais **la fonction ne peut pas le modifier**

Une fois la définition de la fonction terminée, nous revenons au niveau principal pour y définir les deux variables **p** et **q** auxquelles nous attribuons les contenus **15** et **38**.
Ces deux variables définies au niveau principal seront donc des variables globales.
On pourrait croire d'abord que la fonction **essai()** aurait modifié le contenu de la variable globale **p** (puisqu'elle est accessible), mais il n'en est rien : en dehors de la fonction **essai()**, la variable globale **p** conserve sa valeur initiale.