18 - Le Return des paramètres (Programmation)

I - Paramètres d'une fonction

1.1 Fonction : idée abstraite

Une fonction peut être vue comme une boîte noire qui reçoit des données en ENTREE, qui réalise un travail sur ces données puis qui fournit une réponse en SORTIE.

ENTREE(S) ⇒ | FONCTION | ⇒ SORTIE

Exemple d'une fonction capable de recevoir deux informations et de renvoyer un calcul basé sur les informations reçues.

a
⇒ | FONCTION | ⇒ (a+b)*10
b

1.2 Fonction : déclaration en Python

On déclare d'une fonction avec le mot-clé **def**. On peut récupérer les données qu'on envoie en ENTREE lors de l'appel dans des **variables locales** un peu particulière qu'on nomme les **paramètres** de la fonction.

Exemple :

```
1 def f(a, b):
2     return (a+b)*10
```

Ligne 1 : on signale qu'on veut créer une fonction nommée **f** et lui attribuer deux paramètres **a** et **b**.

On notera bien que les paramètres **a** et **b**

1. qu'il s'agit de **variables locales** qui n'existeront que pendant l'exécution réelle de la fonction.
2. que leurs contenus réels ne seront connus que lors de l'appel de la fonction.

1.3 Arguments d'appel et paramètres de fonction

```
1   def f(a, b):
2     return (a+b)*10
3
4   x = f(2, 6)
5   y = f(1, x)
```

Pour comprendre ce que reçoit la fonction lors des appels, il faut comparer la déclaration de la fonction (L1) et les appels (L4 et L5).

```
1   def f(a, b):
4   x = f(2, 6)
```

DEFINITION : Arguments *A comme Appel*

Les données envoyées à la fonction lors de l'appel. Ici, les arguments sont **2** et **6** lors de l'appel L4.

DEFINITION : Paramètres *P comme Prototypage*

Les variables locales qui permettront de stocker les données envoyées à la fonction. Ici, les paramètres sont **a** et **b** (voir la déclaration du prototype L1).

II – Une documentation claire

2-1 Communication programmeur humain - interpréteur Python

Les fonctions que vous écrivez en Python doivent respecter la **syntaxe de Python** de façon à ce que votre texte soit compréhensible par l'interpréteur Python.

Si vous ne suivez pas les règles, l'interpréteur ne vous comprendra pas et va vous le dire en signalant en rouge une **erreur de syntaxe** : votre demande ne respecte pas sa façon de communiquer.

Une **indication en rouge apparaît dans la console Python**. Cela vous permet de localiser et comprendre d'où vient le problème de communication.

```
>>> %Run script_essai.py
def f(a,b)
          ^
SyntaxError: invalid syntax
```

L'interpréteur Python est perdu car on a oublié de placer le : final de la déclaration de la fonction.

2-2 Communication utilisateur humain - concepteur humain

2.2.A - Intérêt de la documentation :

Pour permettre au concepteur de la fonction de communiquer avec l'utilisateur de la fonction, on utilise la documentation de la fonction.

La **documentation sert à savoir comment utiliser le code**. Une sorte de mode d'emploi de votre fonction. Ces informations doivent être suffisantes pour utiliser la fonction sans provoquer d'erreur.

Les délimiteurs de la documentation sont **3 guillemets simples en Python**.

2.2.B - Contenu de la documentation : la documentation doit contenir au minimum :

1. Une courte phrase expliquant à quoi elle sert (ici en L2)
2. Les **préconditions** : les contraintes sur les types et valeurs possibles des paramètres (ici en L4 et L5)
3. Les **postconditions** : une description donnant le type et le contenu de la réponse (ici en L6)

Exemple

```
 1 def moyenne(a, b):
 2     '''Renvoie la moyenne des notes a et b
 3
 4     :: param a(int|float) : nombre dans l'intervalle [ 0 ; 20 ]
 5     :: param b(int|float) : nombre dans l'intervalle [ 0 ; 20 ]
 6     :: return (float): la moyenne des deux notes, dans [ 0.0 ; 20.0 ]
 7
 8     '''
 9     m = (a + b) / 2
10     return m
```

2.2.C - Récupérer la documentation : fonction native **`help()`** depuis la console : **`>>> help(moyenne)`**

2.2.D - Contrat de confiance

Les **préconditions** sont les conditions ou contraintes que l'**utilisateur de la fonction doit respecter.**

Il y a donc une sorte de contrat de confiance entre utilisateur et concepteur :

- L'utilisateur est le garant du respect des préconditions lors des appels
- Le concepteur garantit que la postcondition est alors vérifiée (sous condition du respect des préconditions donc).

Premier cas : l'utilisateur respecte les préconditions

Dans ce cas, l'utilisateur peut concevoir le reste de son programme en considérant que la postcondition est vraie : le concepteur de la fonction s'y est engagé.

Si la fonction donne une mauvaise réponse ou provoque une exception, c'est la **faute du concepteur.**

Deuxième cas : l'utilisateur ne respecte pas les préconditions

Dans ce cas, on ne peut pas tenir compte de la postcondition. La fonction peut :

- fonctionner correctement par hasard, ou
- donner une fausse réponse mais qui possède le bon type, ou
- donner une réponse qui ne possède même pas le bon type, ou
- provoquer une erreur.

En tous cas, c'est clairement la **faute de l'utilisateur** : il a utilisé la fonction hors des clous.

III – Une documentation plus courte : typing

Certaines fonctions ont une documentation qui dépasse en taille le code de la fonction. Pour ces fonctions, on peut passer par une documentation

- moins longue (avantage) mais
- moins précise (désavantage).

```
1 def convertir_sur_10(note:int) -> int:
2     '''Renvoie une note sur 10'''
3     return note // 2
```

- le paramètre note devrait être un integer (**`note:int`**)
- que la fonction renvoie un integer (**`-> int`**).