- Langages et Programmation -

LES FONCTIONS AVEC PYTHON

Plan du chapitre

I. INTRODUCTION

La programmation est l'art d'apprendre à un ordinateur comment accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation sous la forme de **fonctions**.

Les fonctions permettent de décomposer un programme complexe en une série de sous-programmes plus simples. Les fonctions sont en fait des "**blocs de code indépendant réutilisables**" : si nous disposons d'une fonction capable réaliser une tâche bien spécifique, nous pouvons l'appeler plusieurs fois dans notre programme sans avoir à la réécrire systématiquement (on parle de factorisation du code).
Retour au plan

II. FONCTIONS SIMPLES – FONCTIONS PARAMÉTRÉES - PROCÉDURES

Fonction simple

Syntaxe

Pour définir une fonction simple, la syntaxe à respecter est la suivante :

```
def nomFonction ( ) :

        instruction 1
        instruction 2
        instruction 3
              ...

        return résultat
```

corps de la fonction (indentation obligatoire)

Pour nommer votre fonction, vous pouvez choisir le nom que vous voulez, à la condition de ne pas utiliser un nom réservé par le langage, de ne pas utiliser de caractères spéciaux ou accentués (le souligné "_" est autorisé). Une bonne pratique est d'utiliser des noms explicites, de ne pas mettre de majuscule en début de nom d'une fonction (réservé aux **classes)**.

Par exemple pour nommer une fonction qui retourne la valeur absolue d'un nombre :

- abs() → NON, mot réservé à la fonction d'origine dans python
- val abs() → NON, l'espace n'est pas autorisé
- val_abs() → OUI
- valAbs() : OUI

Les parenthèses qui suivent le nom de la fonction sont obligatoires.

Exemple de fonction simple

La fonction suivante, appelée *lanceDes (),* va retourner un nombre entier aléatoire compris entre 1 et 6 inclus :

```
1 from random import *
2
3 # Définition de la fonction
4 def lanceDes() :
5     x=randint(1,6)
6     return x
7
8 # Appel de la fonction et affichage du résultat
9 print(lanceDes())
```

Important : la définition de la fonction doit impérativement précéder son appel, sinon la fonction ne sera pas reconnue.

Remarque : Vous constatez qu'une fonction peut elle-même en appeler une ou plusieurs autre(s). Ici la fonction *lanceDes()* appelle la fonction *randit()*.

Retour au plan

Fonction paramétrée

Nous souhaitons maintenant écrire une fonction nommée *plusPetit* qui va nous renvoyer le plus petit de 2 nombres passés comme **arguments** de la fonction. Dans la définition d'une telle fonction, il faut prévoir 2 variables particulières pour recevoir les arguments transmis. Ces variables s'appellent des **paramètres**.

Syntaxe

On choisit le nom des paramètres en respectant les mêmes règles de syntaxe que d'habitude pour les variables et on place leurs noms entre les parenthèses qui accompagnent la définition de la fonction, séparées par une virgule.

def nomFonction (paramètre1, paramètre2…) :

instruction 1
instruction 2
instruction 3
…

} corps de la fonction (indentation obligatoire)

return résultat

Retour au plan

Exemple de fonction paramétrée

Voici le code qui correspond à l'écriture de la fonction plusPetit :

```
#Définition de la fonction plusPetit
def plusPetit (a,b):    # a et b sont les paramètres de la fonction
    if (a<b) :
        return a
    else :
        return b

# Programme principal
resultat = plusPetit (12,-28)
print("le plus petit nombre est ",resultat)  # 12 et -28 sont les arguments de la fonction
```

Généralement, on utilise des variables comme arguments :

```
#Définition de la fonction plusPetit
def plusPetit (a,b):
    if (a<b) :
        return a
    else :
        return b

# Programme principal
nb1 = float(input("nombre n°1 = "))
nb2 = float(input("nombre n°2 = "))
resultat = plusPetit(nb1,nb2)
print("Parmis les 2 nombres ", nb1, " et ", nb2," le plus prtit nombre est ",resultat)
```

Ici les paramètres *a* et *b* reçoivent respectivement les valeurs des arguments *nb1* et *nb2*.
Remarque : les arguments peuvent être de différents types : booléen, entier, flottant, chaîne de caractère, listes … tout comme les valeurs retournées.

Retour au plan

Arguments facultatifs

Il suffit de définir une valeur par défaut à un argument pour le rendre **facultatif**. Naturellement, cette valeur est écrasée si l'argument est précisé lors de l'appel de la fonction

```
def affiche (x = "aucun argument donné, voici le texte par défaut") :
    "cette fonction affiche le texte passé en argument"
    print(x)

affiche()
affiche("coucou")
```

Retour au plan

Arguments nommés

Pour ne pas être obligé de remplir tous les arguments facultatifs, il est possible de n'en appeler que quelques-uns s'ils sont nommés. Cela permet aussi de rentrer les arguments de la fonction dans un ordre quelconque :

```
def moyenne (note1, note2=10, note3=10 ) :    # note1 : argument obligatoire, note2  et note3 : arguments facultatifs
    "cette fonction calcule la moyenne des 3 arguments"
    print ((note1+note2+note3)/3)

moyenne(note1=20)       # seul l'argument obligatoire est renseigné, les 2 autres prendront la valeur par défaut
moyenne (note2=5, note3=14, note1=18)   # On peut maintenant rentrer les arguments dans un ordre quelconque
```

Retour au plan

Procédure

Si une fonction ne retourne rien, dans ce cas et en toute rigueur, on ne parle plus de fonction mais de **procédure**.
La procédure se termine par **return None** (*None(rien)* représente l'absence de valeur).
Remarque : *return None* n'est pas obligatoire mais vivement conseillé.

Exemple de procédure

```
#Définition de la procédure bonjour()
def bonjour (nom):    # nom est le paramètre de la fonction
    print ("Bonjour ",nom)
    return None

# Programme principal
nom = input("Quel est ton nom ? ")
bonjour(nom)
```

Retour au plan

III. LA PORTÉE DES VARIABLES : VARIABLE LOCALE ET VARIABLE GLOBALE

Variable locale

Considérons le script suivant :

```
def fonction()->None:
    "fonction qui affecte la valeur 'Bonjour' à la variable text"
    text = "Bonjour"
    return None

fonction()
print(text)
```

Lors de l'interprétation du code, on constate que la variable *text* n'est pas reconnue dans le corps du script :

```
  File "E:/Users/Hubert/Mes documents/LCDG/NSI/PgmsPython/
sanstitre0.py", line 7, in <module>
    print(text)

NameError: name 'text' is not defined
```

Cette variable, définie dans le corps de la fonction, ne sera reconnue qu'à l'intérieur de cette fonction, il s'agit d'une **variable locale.**

Retour au plan

Variable globale

Considérons maintenant le programme suivant :

```
def fonction()->None:
    "fonction qui affiche la valeur de la variable text"
    print(text)
    return None

text = "Bonjour"
fonction()
```

Lors de l'interprétation du code, on constate que la variable *text* est bien visible dans le corps de la fonction. Cette variable, définie dans le corps du script, est une **variable globale.**

Retour au plan

Une variable globale peut-elle être modifiée par une fonction ?

Considérons le script suivant. Va-t-il afficher 'Bonjour' ou 'Au revoir' ?

```
def fonction()->None:
    "fonction qui affecte la valeur 'Bonjour' à la variable text"
    text = "Bonjour"
    return None

text = "Au revoir"
fonction()
print(text)
```

Nous avons vu qu'une variable globale est visible à l'intérieur d'une fonction, cependant on remarque ici **que la valeur d'une variable globale ne peut pas être modifiée par une fonction.**

Tout ceci peut paraître bien compliqué au premier abord. Vous comprendrez cependant très vite qu'il est utile que les variables locales restent confinées à l'intérieur du corps des fonctions. Cela signifie que vous pouvez utiliser n'importe quelle fonction sans vous préoccuper le moins du monde du nom des variables utilisées. Ces variables ne pourront pas interférer avec celles que vous avez définies dans le corps de votre script.

Retour au plan

IV. LES RÈGLES DE BON USAGE

Prototyper la fonction

Prototyper une fonction (ou une procédure) consiste à déclarer dans l'entête de définition de la fonction le type des paramètres et le type de la donnée retournée. C'est absolument indispensable dans des langages comme le C, mais pas obligatoire en python. **C'est malgré tout vivement conseillé en python pour la clarification de la fonction au niveau du programmeur, surtout si ce n'est pas lui qui a écrit la fonction.**

```
# Définition de la fonction plusPetit
def PlusPetit (a : float, b:float) -> float:
    if (a<b) :
        return a
    else :
        return b

# Programme principal
nb1 = float(input("nombre n°1 = "))
nb2 = float(input("nombre n°2 = "))
```

Pour une procédure :

```
#Définition de la procédure bonjour()
def bonjour (nom : str ) -> None :
    print ("Bonjour ",nom)
    return None

# Programme principal
nom = input("Quel est ton nom ? ")
bonjour(nom)
```

Retour au plan

Documenter la fonction

L'entête de définition de la fonction peut être suivi d'un commentaire, appelé **docstring,** placé entre des guillemets, voir entre des triples guillemets si le commentaire comporte plusieurs lignes. Ce dernier n'a **aucun rôle fonctionnel** dans le script, mais il est mémorisé dans un **système de documentation interne automatique**, lequel pourra être exploité par certains éditeurs "évolués" comme Spyder.

```
# Définition de la fonction plusPetit
def plusPetit (a : float, b : float) -> float :
    "fonction qui retourne le plus petit nombre"
    if (a<b) :
        return a
    else :
        return b

# Programme principal
nb1 = float(input("nombre n°1 = "))
nb2 = float(input("nombre n°2 = "))

resultat = plusPetit(nb1,nb2)
print("le plus petit nombre est",resultat)
```

Si l'on place la souris sur le nom de la fonction et que l'on effectue un CTRL+I (appel de l'aide), on obtient :

Remarque : l'aide indique également les types des arguments a et b attendus par la fonction, si cela a été renseigné lors du prototypage.

Dans la console python on peut également obtenir l'aide en tapant *help (nomFonction)*

```
In [46]: help (plusPetit)
Help on function plusPetit in module __main__:

plusPetit(a: float, b: float) -> float
    fonction qui retourne le plus petit nombre
```

Retour au plan

Gérer les erreurs

Prenons l'exemple d'une fonction chargée de calculer l'indice de masse corporelle d'une personne.
L'IMC se calcule de la manière suivante : IMC = masse/taille², la masse étant exprimée en kg et la taille en m.

Lorsque l'on définit une nouvelle fonction il est impératif d'analyser les **préconditions sur les arguments** et les **postconditions sur la valeur retournée**.

Analyse des préconditions sur les arguments

Il s'agit d'une condition à vérifier avant l'appel d'une fonction afin de pouvoir en garantir le résultat. On pourrait résumer une précondition ainsi : *« Donne-moi ce que je veux, tu auras ce que tu attends ».*

Pour notre programme, il faut s'assurer que :

- Les valeurs saisies sont impérativement de type entier ou flottant ;
- L'argument masse est compris entre 3 kg et 150 kg ;
- L'argument taille est comprise entre 0.30 m et 2.50 m.

Analyse des postconditions sur la valeur retournée

Il s'agit d'une condition qui doit toujours être vérifiée à la fin de la fonction.
Pour notre programme, il faut vérifier que :

- La valeur retournée est de type flottant ;
- La valeur retournée est comprise entre 10 et 50 (dans le cas contraire, on peut raisonnablement douter de la véracité des arguments saisis. Par exemple masse=149 et taille = 0.50 (arguments dans les limites fixées) donne un IMC de 596 ! évidemment une personne de 50 cm ne peut raisonnablement pas peser 149 kg !)

Ici la documentation de la fonction (voir paragraphe précédent) prend toute son importance pour éviter ce type d'erreur au développeur :

```
def calculIMC (masse : float, taille : float) -> float :
    """ Cette fonction calcule l'indice de masse corporelle d'une personne

    Pré : l'argument masse est un entier ou flottant compris entre 3.0 et 150.0
          l'argument taille est un entier ou flottant compris entre 0.20 et 2.50

    Post : la valeur retounée est un flottant
           la valeur retournée est comprise entre 10.0 et 50.0"""
```

Remarque : plusieurs lignes de commentaires peuvent être placée entre des triples guillemets.

Retour au plan

Les assertions

Python propose un mécanisme **d'assertion** afin de vérifier que les conditions qui doivent être satisfaites le sont effectivement. Une assertion se compose d'une condition (une expression booléenne) suivie d'une virgule et d'une phrase en langue naturelle, sous forme d'une chaine de caractères. L'instruction **assert** teste si sa condition est satisfaite. Si c'est le cas, elle ne fait rien et sinon elle arrête immédiatement l'exécution du programme en affichant éventuellement la phrase qui lui est associée. L'assertion est une aide au développeur et ne doit en aucun cas faire partie du code fonctionnel d'un programme.

```
def calculIMC (masse : float, taille : float) -> float :
    """ Cette fonction calcule l'indice de masse corporelle d'une personne

    Pré : l'argument masse est un entier ou flottant compris entre 3.0 et 150.0
          l'argument taille est un entier ou flottant compris entre 0.20 et 2.50

    Post : la valeur retounée est un flottant
           la valeur retournée est comprise entre 10.0 et 50.0"""

    assert type(masse)==int or type(masse)==float, "l'argument masse doit être de type int ou float"
    assert type(taille)==int or type(taille)==float, "l'argument taille doit être de type int ou float"
    assert masse>3,"masse trop faible"
    assert masse<150,"masse trop élevée"
    assert taille>0.30,"taille trop faible"
    assert taille<2.50,"taille trop grande"

    imc = masse/taille**2
    assert 10<=imc<=50,"valeur d'IMC non plausible, vérifier la véracité des arguments "

    return imc


IMC = calculIMC(84,1.75)
print ("IMC = ",IMC)
```

Il est recommandé de gérer les erreurs à l'aide de test conditionnels if… else… dans le corps de la fonction afin de diminuer le nombre de préconditions sur les arguments.

Retour au plan

V. IMPORTER SES PROPRES FONCTIONS DEPUIS UN MODULE

Lorsqu'un script est exécuté depuis un programme principal, il existe une variable système nommée __*name*__ (avec un double souligné de part et d'autre de ***name***) qui prend systématiquement la valeur *"__main__"* (*main* comme *principal*).
Lorsque du code est exécuté comme à l'intérieur d'un module appelé depuis un programme principal, alors cette variable système __*name*__ prend comme valeur *le nom du module*.

Pour s'en convaincre, reprendre le programme précédent et rajouter une instruction pour afficher la valeur de la variable __name__ :

```
def calculIMC (masse : float, taille : float) -> float :
    """ Cette fonction calcule l'indice de masse corporelle d'une personne

    Pré : l'argument masse est un entier ou flottant compris entre 3.0 et 150.0
          l'argument taille est un entier ou flottant compris entre 0.20 et 2.50

    Post : la valeur retounée est un flottant
           la valeur retournée est comprise entre 10.0 et 50.0"""

    assert type(masse)==int or type(masse)==float, "l'argument masse doit être de type int ou float"
    assert type(taille)==int or type(taille)==float, "l'argument taille doit être de type int ou float"
    assert masse>3,"masse trop faible"
    assert masse<150,"masse trop élevée"
    assert taille>0.30,"taille trop faible"
    assert taille<2.50,"taille trop grande"

    print ("dans le module fonctionNSI, la valeur de la variable __name__ prend la valeur",__name__)

    imc = masse/taille**2
    assert 10<=imc<=50,"valeur d'IMC non plausible, vérifier la véracité des arguments "

    return imc
```

Enregistrer ce fichier sous le nom *"fonctionsNSI"*

Écrire maintenant un programme principal qui appelle la fonction *calculIMC* dans le module *fonctionNSI*, en rajoutant également une instruction pour afficher la valeur de la variable *__name__ :*

```
1 from fonctionsNSI import *
2
3
4 IMC = calculIMC(84,1.75)
5 print ("IMC = ",IMC)
6
7 print ("dans le pgm principal, la variable __name__ prend la valeur : ", __name__)
```

L'instruction *from fonctionsNSI import calculIMC* permet d'importer la fonction *calculIMC* du module *fonctionsNSI.*

L'instruction *from fonctionsNSI import ** permet d'importer toutes les fonctions du modules *fonctionNSI* s'il y en a plusieurs.

Exécutez maintenant ce script, on peut alors vérifier le bon fonctionnement du module ainsi que les valeurs prises par la variable *__name__*.

```
In [45]: runfile('E:/Users/Hubert/Mes documents/LCDG/NSI/PgmsPython/
essais.py', wdir='E:/Users/Hubert/Mes documents/LCDG/NSI/PgmsPython')
Reloaded modules: fonctionsNSI
dans le module fonctionNSI, la valeur de la variable __name__ prend
la valeur fonctionsNSI
IMC =  27.428571428571427
dans le pgm principal, la variable __name__ prend la valeur :
__main__
```

Il est possible d'exécuter directement un module importable comme un script. Pour cela, à la fin du module *fonctionsNSI*, rajouter les lignes de code suivantes :

```
24 if __name__ == "__main__" :
25     IMC = calculIMC(84,1.75)
26     print (IMC)
```

Avec ce test conditionnel, le code du corps du test n'est pas exécuté si le fichier est importé, cependant il le sera si le module est exécuté comme un fichier "__main__".

Retour au plan

VI. EXERCICES

Dans tous les exercices, les fonctions seront prototypées, documentées et des assertions seront réalisées suivant le cahier des charges.

Exercice n°1 :
Écrire une fonction nommée *paire* qui prend en argument un nombre entier strictement positif et retourne *True* si ce nombre est paire, *False* sinon.

Exercice n°2 :
Écrire une fonction nommée *divisible* qui prend en argument deux nombres entiers et retourne *True* si le plus grand des 2 nombres est divisible par le plus petit, *False* sinon.

Exercice n°3 :
Écrire une fonction nommée *nbCaract* qui prend en 1er argument une chaîne de caractères, en 2nd argument un caractère et retourne le nombre de fois que le caractère apparait dans la chaine. Le 2nd argument est facultatif, s'il n'est pas indiqué il sera considéré par la fonction comme un espace. Si le caractère n'est pas présent dans la chaine, la fonction retourne 0.

Exercice n°4 :
Écrire une fonction nommée *inverse* qui prend en argument une chaine de caractères et retourne la chaine de caractères inversées.

Retour au plan