UNIVERSITE JOSEPH KI-ZERBO/IFOAD

INTRODUCTOIN A PYTHON

KYELEM YACOUBA

Doctorant en Informatique

Université Joseph Ki-Zerbo

Laboratoire de Mathématiques et d'Informatique (LAMI)

Contact : kyelemy@yahoo.fr/71025937

Plan du cours

Chapitre 3 : Les fonctions

I. Définition

Une fonction est un bloc de code nommé. Une fonction correspond à un ensemble d'instructions créées pour effectuer une tâche précise, regroupées ensemble et qu'on va pouvoir exécuter autant de fois qu'on le souhaite en "l'appelant" avec son nom. Notez "qu'appeler" une fonction signifie exécuter les instructions qu'elle contient.

L'intérêt principal des fonctions se situe dans le fait qu'on va pouvoir appeler une fonction et donc exécuter les instructions qu'elle contient autant de fois qu'on le souhaite, ce qui constitue au final un gain de temps conséquent pour le développement d'un programme et ce qui nous permet de créer un code beaucoup plus clair.

Il existe deux grands "types" de fonctions en Python : les fonctions prédéfinies et les fonctions créées par l'utilisateur. Les fonctions prédéfinis ont été vu dans le chapitre 1 nous verrons dans la suite les fonctions définies par l'utilisateur.

II. Déclaration d'une fonction

Pour définir une nouvelle fonction en Python, nous allons utiliser le mot clef def qui sert à introduire une définition de fonction. Ce mot clef doit être suivi du nom de la fonction, d'une paire de parenthèses au sein desquelles on pourra fournir une liste de paramètres ou arguments et de : pour terminer la ligne comme ceci def ma_fonction():.

Le nom d'une fonction Python doit respecter les normes usuelles concernant les noms :

- ✓ les noms de fonction doivent commencer par une lettre ou un underscore et ne contenir que des caractères alphanumériques classiques (pas d'accent ni de cédille ni aucun caractère spécial).
- ✓ les noms de fonctions sont sensibles à la casse en Python.

Nous allons ensuite placer la liste des différentes instructions de notre fonction à la ligne suivant sa définition et en les indentant par rapport à la définition afin que Python comprenne que ces instructions appartiennent à notre fonction.

Exemple

```
>>>def toto():
    print("La fonction s'exécute")
```

Il est possible de définir une fonction à l'intérieur d'une autre fonction. Dans ce cas, la fonction n'est visible qu'à l'intérieur de la fonction principale et ne peut être utilisée dans le champ global:

```
def f(x):
    from math import exp
```

```
    def g(y): return y**2+1
    return exp(g(x)) / g(x)
print(f(1), g(2))
```

Dans l'exemple, l'affichage de g(2) renvoie une erreur. Cela peut être très pratique pour structure le corps d'une fonction complexe sans avoir à polluer le niveau global avec beaucoup de fonctions.

III. Exécution d'une fonction

Après la définition Il suffit ensuite d'utiliser le nom de la fonction suivie de parenthèses pour exécuter le code qu'elle contient :

Exemple :

>>>def toto():

print("La fonction s'exécute")

>>>toto()

Le code contenu dans le bloc de notre fonction est alors exécuté, c'est pourquoi nous voyons s'afficher les messages passés à print.
Tout le code de la fonction sera à nouveau exécuté à chaque nouvel appel, chaque appel étant indépendant des autres.

IV. Retours de fonction

>>>def toto():

print("La fonction s'exécute")

la fonction ci-dessus permet d'afficher des valeurs mais ne renvoient rien (ou plutôt renvoient **None**).

On retourne une valeur dans une fonction grâce au mot clé return. Ce mot clé a pour effet de mettre fin à l'exécution de la fonction.

Exemple :

>>>def somme(a, b):

return a + b

Une fonction n'est pas limitée à un seul return et il est ainsi possible d'en avoir plusieurs pour Contrôler le flux d'exécution. L'exécution de la fonction s'arrêtera au premier return rencontré, renvoyant la valeur associée à l'expression de ce return.

```python
def division(a, b):
    if b == 0:
        return 0
    return a / b
```

La valeur retournée par une fonction peut être récupérée dans une variable lors de l'appel de la fonction.

V. Les fonctions paramétrées en python

Le paramètre est une variable définie dans la fonction qui recevra une valeur lors de chaque appel. Cette valeur pourra être de tout type, suivant ce qui est fourni en argument.
Les noms des paramètres sont inscrits lors de la définition de la fonction, entre les parenthèses qui suivent son nom. S'il y a plusieurs paramètres, ils doivent être séparés par des virgules.

Exemple

```python
def table_multiplication(n):
    for i in range(1, 11):
        print(n, '×', i, '=', n*i)
```

```python
def hello(firstname, lastname):
    print('Hello', firstname, lastname, '!')
```

Lors de l'appel de la fonction, on utilisera les arguments pour donner leurs valeurs aux paramètres.
On précise ainsi les valeurs dans les parenthèses qui suivent le nom de la fonction.

Exemple : table_multiplication(3)

Le comportement est le même pour les fonctions à plusieurs paramètres, les valeurs leur sont attribuées dans l'ordre des arguments : le premier paramètre prend la valeur du premier argument, etc. Chaque argument correspond ainsi à un paramètre (et inversement).

Exemple : **hello('tata','toto')**

Arguments optionnels: il est possible de mettre des arguments optionnels à condition de leur donner une valeur par défaut et ils doivent nécessairement être placés après les arguments obligatoires:

Exemple :

def division(a,b,c=0) : return a/b

Arguments nommés: lorsqu'il y a des paramètres optionnels, on peut ne passer que l'argument dont on veut attribuer une valeur différente de la valeur par défaut, sans avoir à respecter l'ordre:

Liste extensible d'arguments non nommés: Python permet à une fonction d'accepter une liste extensible d'arguments, c'est-à-dire que le nombre d'arguments n'est pas défini à l'avance. Il faudra dans ce cas veiller à ce que la fonction gère correctement cette liste quelconque d'arguments. La syntaxe est la suivante, avec une * devant le nom de ce qui sera un tuple

contenant la liste des paramètres (on aime bien utiliser 'args' mais on peut appeler ce tuple autrement):

```
def f(*args):
    print(args)
    if len(args) == 3:
        a,b,c = args[0],args[1],args[2]
        return a+b+c
    else:
        return None
f(1,2,3)
f(3)
```

Liste extensible d'arguments nommés: il est en fait plus judicieux d'utiliser une liste de paramètres nommés extensible. La syntaxe est la suivante, avec cette fois une double étoile ** et 'kwargs' n'est alors plus un tuple mais un dictionnaire associant nom et valeur des variables à transmettre:

```
def f(**kwargs):
    print(str(kwargs))
    if len(kwargs) == 3:
        a,b,c = kwargs['a'],kwargs['b'],kwargs['c']
        return a+b**2+c**3
    else:
        return None
print(f(a=1,b=2,c=3), f(c=1,b=2,a=3))
print(f(c=2))
```

On peut mixer tous ces mécanismes de passage d'arguments mais en respectant l'ordre suivant: arguments obligatoires, optionnels, extensible non nommés puis extensible nommés soit:

```
def g(a,b=2,*args,**kwargs):
    print(a,b,args,kwargs)
g(1,2,3,4,5,x=3)
```

VI. Variables dans les fonctions

1. Variable locale

Cette variable est liée à l'exécution de la fonction. Cela signifie qu'elle n'existe pour l'interpréteur qu'au moment de l'exécution de la fonction. Dès que la fonction se termine, la variable et sa valeur ne sont plus accessibles. On dit que la portée de la variable est limitée à la fonction. Donc cette variable n'existe pas dans le reste du programme.

```
def compteur(stop):
    i = 0
    while i < stop:
        print(i)
        i = i + 1
a = 5
compteur(a)
```

La variable i est local à la fonction. Par contre nous pouvons rendre i global en le déclarant global.

```
def compteur(stop):
    global i = 0
    while i < stop:
        print(i)
        i = i + 1

a = 5
compteur(a)
```

2. Variable globale

Une variable qui est déclarée dans le fichier principal est aussi appelée variable globale car elle existe dans la mémoire jusqu'à la fin de l'exécution du programme.

Exemple :

```
def compteur(stop):
    i = 0
    while i < stop:
        print(i)
        i = i + 1

a = 5
compteur(a)
```

la variable a est une variable global.

VII. Appel récursif de fonction

Une fonction peut bien évidemment appeler une autre fonction dans le corps de son traitement. Mais une fonction peut aussi s'appeler elle-même. On parle alors d'appel récursif. Un appel récursif implique toujours une condition d'arrêt sinon la fonction va s'appeler sans fin ou plus exactement il y aura une fin appelée débordement de pile d'appel qui fait échouer le programme avec une erreur de type.

Une condition d'arrêt empêche la fonction de s'appeler elle-même à l'infini. Un exemple classique d'appel récursif et le calcul de la factorielle d'un nombre.

```
def factorielle(n):
    if n > 0:
        return n * factorielle(n-1)
    else:
        return 1
```

Références

https://gayerie.dev/docs/python/python3/

https://www.pierre-giraud.com/python-apprendre-programmer-cours/introduction-fonction/

Python: Fonctions - Wiki Cours (u-psud.fr)

Cours Matthieu Boileau & Vincent Legoll Apprendre Python pour les sciences CNRS - Université de Strasbourg.

Cours de Python Introduction à la programmation Python pour la biologie Patrick Fuchs et Pierre Poulain https://python.sdv.univ-paris-diderot.fr/ .