Initiation à Python - leçon 3.2.4

s1
Dans cette séquence, nous allons découvrir les fonctions.

---
s2

Les langages de programmation offrent souvent la possibilité d'isoler des parties de code que l'on souhaite utiliser plusieurs fois dans le même programme sans les réécrire. On les appelle des fonctions.

En python, une fonction est également un objet.

---
s3

Une fonction est un bloc d'instructions qui a reçu un nom, dont le fonctionnement dépend d'un certain nombre de paramètres qui sont passés à la fonction (on les appelle aussi les arguments de la fonction) et qui renvoie un résultat (au moyen de l'instruction return). Ce renvoi d'informations peut ne pas être explicitement précisé, dans ce cas la fonction renvoie l'objet Null.

Dans Python, la construction est la suivante : le mot-clé def signifie que l'on définit une fonction il est suivi du nom de la fonction. Ce nom de fonction est en fait une variable qui pointe vers l'objet fonction.
Ce nom est suivi de parenthèses entourant les arguments passés à la fonction et le symbole "deux points". Si la fonction n'a pas besoin d'arguments, il faut quand même écrire les deux parenthèses. Les arguments représentent des objets qui sont passés à la fonction par référence. Nous verrons cela plus tard.

La partie indentée constitue le corps de la fonction. La plupart du temps, on trouve l'instruction return suivie de l'objet retourné par la fonction.

Voyons cela sur un exemple.

Dans IDLE, ouvrons un nouveau fichier, écrivons ce code et exécutons-le.

Ce code signifie que l'on définit une fonction reste avec deux arguments a et b. la fonction effectue une division entière de a par b, calcule le reste et le renvoie.

Dans la suite de ce petit programme, on déclare une variable r comme étant le résultat de la fonction reste pour les arguments 5 et 2. On trouve bien entendu 1.

Attention, l'ordre dans lequel apparaissent les paramètres dans la définition de la fonction est important. Les valeurs passées en paramètre lors de l'appel doivent respecter cet ordre.

Si dans le terminal on écrit g = divise, on définit une nouvelle variable g qui pointe vers le même objet fonction.

faire g(2,6) (dans le terminal - on vient d'exécuter le fichier donc f chargée en mémoire)

Que se passe-t-il si l'on demande l'exécution d'une fonction avant de la définir ?
Python est moins permissif que d'autres langages. Il va afficher une erreur.
Voici un autre exemple dans lequel la fonction renvoie un objet de type liste.

Je vous propose de reproduire tous ces exemples.

---
s4

On peut omettre l'instruction return si la fonction ne renvoie rien.

C'est le cas dans cet exemple élémentaire. La fonction affiche remplit son rôle - afficher la valeur qui lui est transmise. On peut déclarer une variable y comme étant le résultat retourné par la fonction affiche, même si aucune instruction return n'est écrite. En revanche, le type de y est NoneType. Le type NoneType dont la seule valeur est None est le type Python pour représenter le "rien", l'absence de valeur.

Tant que nous y sommes, précisons qu'il peut y avoir plusieurs instructions return dans une fonction (évidemment on ne sortira réellement de la fonction qu'en passant par l'une et l'une seulement de ces instructions). L'exemple qui suit illustre ce cas.

Je vous propose de reproduire cet exemple.

---
s5

Un mot sur le polymorphisme.

Une fonction est dite polymorphe lorsqu'elle reçoit des arguments qui peuvent être de type différent lors de différents appels de cette fonction. Prenons un exemple.

On crée un nouveau fichier et on y définit une fonction qui reçoit a et b en arguments. On sauve et réinitialise l'interpréteur.
On appelle cette fonction avec deux entiers, puis avec deux flottants.
On constate que la fonction s'exécute parfaitement, y compris avec deux chaînes et avec deux listes.
C'est ce qu'on appelle le polymorphisme : la fonction f s'adapte en quelque sorte au type d'objet passé en paramètre. Il y a une contrainte : les opérations doivent être compatibles avec les objets manipulés.

Je vous propose de reproduire cet exemple.

---
s6

Parlons des arguments par défaut.
Un paramètre d'une fonction peut avoir une valeur par défaut.
Une fonction peut accepter un nombre quelconque de paramètres. On peut donner une valeur par défaut à un ou plusieurs arguments d'une fonction.
Si un argument reçoit une valeur par défaut, il peut donc être omis lors de l'appel de la fonction.

Dans l'exemple suivant, on dispose d'une fonction avec deux arguments dont l'un - delta - prend la valeur 1.

Si l'on appelle la fonction incr avec les deux arguments 20 et 3, delta prend la valeur 3.
Si l'on appelle la fonction incr avec un seul argument 20, delta prend la valeur par défaut 1.

Pour qu'il n'y ait aucune ambiguïté, les arguments avec valeurs par défaut (dans la définition de la fonction) doivent suivre ceux qui n'en ont pas. En effet, comment Python peut-il se retrouver avec une définition comme celle présentée ? D'ailleurs, l'interpréteur vous signalera que l'ordre des paramètres est erroné.

Nous l'avons dit plus haut, l'ordre dans lequel apparaissent les paramètres dans la définition de la fonction est important. Les valeurs passées en paramètre lors de l'appel doivent respecter cet ordre. Les arguments sont dits positionnels. On peut cependant contourner cette contrainte en nommant les arguments lors de l'appel à la fonction, comme dans l'exemple présenté.

---
s7

Il est possible (et courant) d'emboîter des définitions de fonctions. Dans l'exemple présenté, nous définissons une fonction f avec un paramètre a et à l'intérieur de cette fonction f, nous définissons une fonction g avec un paramètre b.

A la suite et hors de la définition de g, mais toujours à l'intérieur de la définition de la fonction f, on fait un appel à g avec le paramètre passé à f.

Cet exemple d'emboîtement de fonctions nous permet de mettre en évidence la notion de bloc de code. La définition de la fonction g correspond à un bloc de code ; ce bloc de code est inclus dans le bloc de code correspondant à la définition de la fonction f ; ce dernier étant inclus dans le bloc de code du programme principal qui n'est autre que le fichier Python.

Je vous propose de reproduire cet exemple.

Cet exemple de fonctions imbriquées nous permet d'introduire le concept d'espace de nom. Quand une fonction f est appelée (et que débute son évaluation), elle crée un nouvel espace de noms : cela signifie que les variables qui sont créées dans cette fonction ne sont visibles qu'à l'intérieur de celle-ci - on dit qu'elles sont locales. Notre fonction f peut cependant continuer à accéder aux variables de l'espace global. De même, si la fonction f contient la définition d'une fonction g, celle-ci crée un espace de noms inclus dans celui de f.

Ainsi, f appartient à l'espace de nom du bloc principal ; a et g appartiennent à l'espace de nom de la fonction f. Pour s'en convaincre, il suffit par exemple à la fin de la fonction f d'afficher les variables locales grâce à la méthode locals().
Enfin, b appartient à l'espace de nom de la fonction g.

Cette représentation va être très utile pour comprendre le concept de scope.

---
s8

Nous allons découvrir à présent la notion de scope, en français on parle de contexte. Ce contexte définit ce qu'on appelle la portée d'une variable. Il faut comprendre le mot portée comme une distance. En d'autres termes, la portée d'une variable correspond aux endroits du programme où elle reste définie. Nous allons voir que la portée d'une variable dépend essentiellement de l'endroit du programme - plus précisément du bloc de code - où la variable est définie.

Voyons cela sur un exemple. On reprend l'exemple précédent avec les deux fonctions imbriquées f et g, cette fois sans leurs paramètres afin de simplifier le propos.

Premier exercice : tout au début du fichier Python, on définit une variable i. Cette variable est dans l'espace de nom du bloc principal ; c'est donc une variable locale par rapport au bloc principal ; mais elle est également globale - car le bloc principal est le fichier Python - et à ce titre sera définie dans tous les blocs et sous-blocs contenus dans le programme. La portée de i est représentée par le rectangle violet.

Deuxième exercice : On enlève ce qui concerne i pour plus de clarté. Tout au début de la fonction f, on définit une variable j. Cette variable appartient à l'espace de nom de la fonction f ; elle est appelée variable locale pour la fonction f ; elle sera définie bien sûr dans f, mais également dans tous les blocs et sous-blocs contenus dans f. En revanche, si l'on cherche à utiliser j dans le bloc principal, on provoquera une erreur. La portée de j est représentée par le rectangle violet.

Troisième exercice : tout au début du fichier Python, on définit une variable i. Tout au début de la fonction g, on définit à nouveau une variable i. Que se passe-t-il si l'on cherche à afficher i dans g ? De quel i s'agit-il ?
De manière générale, pour ces questions de scope, Python respecte une règle très simple : la règle LEG. LEG signifie que lorsqu'une variable est utilisée, Python va d'abord chercher cette variable dans le scope ou contexte local. S'il la trouve, comme ici, il l'affiche ; s'il ne la trouve pas, il va chercher dans le contexte englobant, c'est-à-dire dans la fonction f ; s'il ne la trouve pas, il va continuer à chercher dans les contextes de niveau supérieur, et ce jusqu'au niveau global, celui du programme principal. C'est ce qui se passait dans le premier exemple où i était défini seulement au niveau global.

Dernière question que se passe-t-il si l'on cherche à afficher i dans le programme principal ? Python applique la règle LEG. Lorsque l'on cherche à afficher i à la fin du programme principal, on se trouve dans le contexte du programme principal. Python cherche donc i dans ce contexte, il trouve et affiche la valeur 1. Ceci nous permet de vérifier que la redéfinition de i dans le contexte de la fonction g n'a pas modifié la valeur de la variable i du contexte principal. La portée de la variable i définie dans la fonction g est représentée par le rectangle violet.

Cette notion de scope ou contexte est très importante. À chaque fois que vous utilisez une variable dans une instruction Python, la règle LEG s'applique. On retiendra que la recherche d'une variable se fait du scope local vers des scopes plus globaux

Je vous propose de reproduire les exemples présentés ici.

---
s9

Les exemples qui vont suivre sont en quelque sorte un bilan de cette leçon sur les fonctions. Nous allons comprendre ce que signifie le fait que les paramètres de la fonction sont passés en référence ; ceci nous amènera à comprendre le fonctionnement des variables locales ; nous reparlerons à cette occasion de références partagées ; et nous verrons qu'elle peut interférer avec la notion de scope ; nous introduirons alors la notion de variable globale.

Je vous propose de les reproduire un par un.

Dans ce premier exemple, on définit une variable a qui appartient à l'espace de nommage du bloc principal ; c'est une variable globale ; a référence l'objet 1 ; on définit également une variable f de type fonction - le nom de l'argument est appelé x ; ensuite on exécute f(a) ; que se passe-t-il ? x est le nom de l'argument de f, au moment de l'exécution de l'instruction f(a) - a référençant l'objet 1 - on crée une variable x qui appartient à l'espace de nommage de f et qui référencera le même objet 1 ; c'est ce qui se passe lorsque l'on dit que les paramètres sont passés en référence ; ensuite Python évalue l'expression x+3, crée un objet 4 et le fait référencer par la variable x ; ceci supprime le référencement précédent ; ensuite on affiche l'objet référencé par x ; puis, on sort de la fonction f, ce qui supprime toutes les variables locales, ferme l'espace de noms de f ; rien n'est retourné.

Dans ce deuxième exemple, on définit encore une variable a qui référence l'objet 1 ; on définit toujours une variable f de type fonction ; ensuite on exécute f(a) ; que se passe-t-il ? x est le nom de l'argument de f, on crée donc une variable x qui appartient à l'espace de nommage de f et qui référencera le même objet 1 ; ensuite Python évalue l'expression x+3, crée un objet 4 et le fait référencer par une variable locale a ; on affiche l'objet référencé par la variable a ; ensuite on sort de la fonction f ; on supprime donc toutes les variables locales ; rien n'est retourné ; à nouveau on affiche la valeur de a, il n'y a d'ambiguïté : on se trouve dans l'espace de variables associé au bloc principal ; c'est la variable globale a = 1 dont il s'agit.

Dans ce troisième exemple, on définit encore une variable a qui référence l'objet 1 ; on définit toujours une variable f de type fonction ; ensuite on exécute f(a) ; que se passe-t-il ? x est le nom de l'argument de f, on crée donc une variable x qui appartient à l'espace de nommage de f ; à nouveau cette variable référence l'objet 1 ; Python cherche à afficher l'objet associé à une variable a : cette variable n'existe pas dans le contexte local - l'espace de noms de la fonction f - Python va donc chercher dans le contexte englobant, celui du bloc principal ; on trouve une variable a ; on affiche l'objet 1. On sort de la fonction ; on supprime donc toutes les variables locales : c'est-à-dire x.

Ce quatrième exemple est très similaire au précédent : au moment de l'exécution, dans la fonction f, Python cherche à afficher l'objet associé à une variable a : cette variable n'existe pas dans le contexte local Python va donc afficher l'objet associé à la variable globale a ; ensuite Python évalue l'expression x+3 et cherche à l'affecter à une variable locale a ; or, dans l'instruction précédente, a était considéré comme la variable globale ; pour Python, il y a ambiguïté entre ces deux définitions ; un message d'erreur est affiché.

Le cinquième exemple montre comment lever cette ambiguïté. Si on souhaite utiliser depuis l'intérieur d'une fonction une variable globale (non seulement pour la lire, mais pour la modifier), il faut le préciser (au début de la fonction) par la directive global ; après que Python a évalué l'expression x+3, il l'affectera sans problème à la variable globale a. Ensuite, on sort de la fonction ; on supprime donc toutes les variables locales : c'est-à-dire x. On affiche la variable globale a, qui ici a donc été modifiée.

Une remarque : pour éviter toute ambiguïté, la directive global doit être placée AVANT une quelconque déclaration (cette déclaration étant comprise comme la création d'une variable locale).

On pourrait croire que cette directive global est très pratique puisqu'elle permet d'accéder au scope. Cependant, son utilisation est fortement déconseillée (pour ne pas dire interdite) dans les bonnes pratiques de programmation, car elle nuit à la clarté du programme et se révèle dangereuse.

Si l'on désire cependant modifier une variable globale par une fonction, on peut privilégier la façon de faire suivante. On définit encore une variable a qui référence l'objet 1 ; on définit toujours une variable f de type fonction ; ensuite on exécute f(a) ; que se passe-t-il ? x est le nom de l'argument de f, on crée donc une variable x qui appartient à l'espace de nommage de f et qui référencera le même objet 1 ; ensuite Python évalue l'expression x+3, crée un objet 4 et le fait référencer par la variable locale x ; on sort de la fonction f en retournant l'objet référencé par la variable x ; l'objet retourné est affecté à la variable globale a, ce qui supprime le référencement précédent.

Je vous demande de reproduire ces exercices qui devraient vous permettre de bien comprendre comment fonctionne Python du point de vue des fonctions et des variables et ainsi vous éviter de commettre des erreurs.

---
s10

Enfin, voici un exemple qui fait intervenir des listes et qui ne doit pas nous surprendre si l'on se rappelle que les listes sont des objets mutables. Rappelez-vous ! Dire qu'un objet est mutable, c'est dire qu'on peut en modifier (voire en supprimer) un (ou plusieurs) élément(s), sans pour autant créer une nouvelle référence vers l'objet ainsi modifié (l'adresse du début de l'objet reste inchangée).

On définit une variable L qui référence un objet de type liste - c'est un objet mutable - qui pointe vers les objets 1 et 2 ; on définit toujours une variable f qui référence un objet de type fonction qui prend la variable liste en argument ; ensuite on exécute f(L) ; que se passe-t-il ? liste est le nom de l'argument de f, on crée donc une variable liste qui appartient à l'espace de nommage de f et qui référence l'objet liste 1,2 ; à présent, on exécute liste.append(4) ; on prend la variable locale liste et on ajoute l'élément 4 à l'objet liste ; on crée donc un objet 4, on modifie l'objet liste et on fait pointer son dernier indice vers l'objet 4 ; à la ligne suivante, on crée un objet liste 1,2,3 et on le fait référencer par la variable locale liste, ce qui supprime le référencement précédent.
On sort de la fonction f ce qui supprime l'espace de noms de la fonction f.

On affiche l’objet L, ce qui donne 1, 2, 4. L’objet liste L mutable a été transformé par la fonction f.

Pour terminer cette leçon, je vous propose de reproduire cet exemple.