W4-S8-C1-passage-arguments

December 15, 2014

1 Passage d'arguments

1.1 Complément - niveau intermédiaire

1.1.1 Motivation

Jusqu'ici nous avons développé le modèle simple qu'on trouve dans tous les langages de programmation, à savoir qu'une fonction a un nombre fixe, supposé connu, d'arguments. Ce modèle a cependant quelques limitations; les mécanismes de passage d'arguments que propose python, et que nous venons de voir dans les vidéos, visent à lever ces limitations.

Voyons de quelles limitations il s'agit.

1.1.2 Nombre d'arguments non connu à l'avance

Ou encore : introduction à la forme `*arguments` Pour prendre un exemple aussi simple que possible, imaginons que `print` est comme dans python-3, non plus une instruction, mais une fonction. On peut faire cela en python-2 en important le module `__future__` qui permet d'utiliser des fonctionnalités disponibles dans des versions ultérieures de python. Dans le cas qui nous intéresse, l'import ci-dessous permet de remplacer l'instruction `print` par une fonction `print` qui se comporte comme dans python-3.

```
In []: from __future__ import print_function

       print("nous", "utilisons", "print", "comme", "une", "fonction")
```

Imaginons maintenant que nous voulons implémenter une variante de `print`, c'est-à-dire une fonction `error`, qui se comporte exactement comme `print` sauf qu'elle ajoute en début de ligne une balise `ERROR`.

Se posent alors deux problèmes. * D'une part il nous faut un moyen de spécifier que notre fonction prend un nombre quelconque d'arguments. * D'autre part il faut une syntaxe pour repasser tous ces arguments à la fonction `print`.

On peut faire tout cela avec la notation en `*` comme ceci

```
In []: # accepter n'importe quel nombre d'arguments
       def error(*print_args):
           # et les repasser à l'identique à print en plus de la balise
           print('ERROR', *print_args)

       # on peut alors l'utiliser comme ceci
       error("problème", "dans", "la", "fonction", "foo")
       # ou même sans argument
       error()
```

1.1.3 Légère variation

Pour sophistiquer un peu cet exemple, on veut maintenant imposer à la fonction erreur qu'elle reçoive un argument obligatoire de type entier qui représente un code d'erreur, plus à nouveau un nombre quelconque d'arguments pour `print`.

Pour cela, on peut créer une signature qui va mélanger un argument traditionnel en première position, qui sera obligatoire à l'appel, et le tuple des arguments pour `print`, comme ceci

```
In []: def error(error_code, *print_args):
           # ici on peut imaginer une logique qui retrouve un message
           # d'erreur plus complet attaché au code d'erreur
           message = "message d'erreur pour {}".format(error_code)
           print("ERROR", message, '--', *print_args)

       # que l'on peut à présent appeler comme ceci
       error(100, "un", "petit", "souci")
```

Remarquons que maintenant la fonction `error` ne peut plus être appelée sans argument, puisqu'on a mentionné un paramètre obligatoire `error_code`.

1.1.4 Ajout de fonctionnalités

Ou encore : la forme `argument=valeur_par_defaut` Nous envisageons à présent le cas - tout à fait indépendant de ce qui précède - où vous avez écrit une librairie graphique, dans laquelle vous exposez une fonction `ligne` définie comme suit. Évidemment pour garder le code simple, nous imprimons seulement les coordonnées du segment; souvenez vous que ce notebook utilise `print_function`:

```
In []: # première interface pour dessiner une ligne
       def ligne(x1, y1, x2, y2):
           "dessine la ligne (x1,y1) -> (x2,y2)"
           # restons simple
           print("la ligne ({},{}) -> ({},{})".format(x1, y1, x2, y2))
```

Vous publiez cette librairie en version 1, vous avez des utilisateurs; et quelque temps plus tard vous écrivez une version 2 qui prend en compte la couleur. Ce qui vous conduit à ajouter un paramètre pour `ligne`.

Si vous le faites en déclarant

```
def ligne(x1, y1, x2, y2, couleur):
    ...
```

alors tous les utilisateurs de la version 1 vont devoir changer leur code - pour rester à fonctionnalité égale - en ajoutant un cinquième argument 'noir' à leurs appels à `ligne`.

Vous pouvez éviter cet inconvénient en définissant une deuxième version de `ligne` comme ceci

```
In []: # deuxième interface pour dessiner une ligne
       def ligne(x1, y1, x2, y2, couleur="noir"):
           "dessine la ligne (x1,y1) -> (x2,y2) dans la couleur spécifiée"
           # restons simple
           print("la ligne ({},{}) -> ({},{}) en {}".\
                 format(x1, y1, x2, y2, couleur))
```

Avec cette nouvelle définition, on peut aussi bien

```
In []: # faire fonctionner du vieux code sans le modifier
       ligne(0, 0, 100, 100)
       # ou bien tirer profit du nouveau trait
       ligne(0, 100, 100, 0, 'rouge')
```

Les paramètres par défaut sont très utiles Notez bien que ce genre de situation peut tout aussi bien se produire sans que vous ne publiiez de librairie, à l'intérieur d'une seule application. Par exemple, cette situation peut se produire si vous avez un jour à ajouter un argument à une fonction parce qu'elle doit faire face à de nouvelles situations imprévues, mais que vous n'avez pas le temps de modifier tout le code.

Ou encore plus simplement, vous pouvez choisir d'utiliser ce passage de paramètres dès le début de la conception. Si vous implémentez par exemple une fonction `ligne` dans un environnement réaliste, il vous faut une interface qui précise: les points concernés, la couleur du trait, l'épaisseur du trait, le style du trait, le niveau de transparence, etc... Et il n'est vraiment pas utile que tous les appels à `ligne` reprécisent tout ceci intégralement, aussi une bonne partie de ces paramètres seront très constructivement déclarés avec une valeur par défaut.

1.2 Complément - niveau avancé

1.2.1 Écrire un wrapper

Ou encore : la forme `**keywords` La notion de *wrapper* - emballage, en anglais - est très répandue en informatique, et consiste, à partir d'un morceau de code souche existant (fonction ou classe) à définir une variante qui se comporte comme la souche, mais avec quelques légères différences.

La fonction `error` était déjà un premier exemple de *wrapper*. Maintenant nous voulons définir un *wrapper* `ligne_rouge`, qui sous-traite à la fonction `ligne` mais toujours avec la couleur rouge.

Maintenant que l'on a injecté la notion de paramètre par défaut dans le système de signature des fonctions, se repose la question de savoir comment passer à l'identique les arguments de `ligne_rouge` à `ligne`.

Évidemment, une première option consiste à regarder la signature de `ligne`:

```
def ligne(x1, y1, x2, y2, couleur="noir")
```

Et à en déduire une implémentation de `ligne_rouge` comme ceci

```
In []: # la version naïve - non conseillée - de ligne_rouge
       def ligne_rouge(x1, y1, x2, y2):
           return ligne(x1, y1, x2, y2, couleur='rouge')

       ligne_rouge(0, 0, 100, 100)
```

Toutefois, avec cette implémentation, si la signature de `ligne` venait à changer, on serait vraisemblablement amené à changer celle de `ligne_rouge`, sauf à perdre en fonctionnalité. Imaginons en effet que `ligne` devienne dans une version suivante

```
In []: # on ajoute encore une fonctionnalité à la fonction ligne
       def ligne(x1, y1, x2, y2, couleur="noir", epaisseur=2):
           print("la ligne ({},{}) -> ({},{}) en {} - ep. {}".\
                   format(x1, y1, x2, y2, couleur, epaisseur))
```

Alors le wrapper ne nous permet plus de profiter de la nouvelle fonctionnalité. De manière générale, on cherche au maximum à se prémunir contre de telles dépendances. Aussi, il est de beaucoup préférable d'implémenter `ligne_rouge` comme suit, où vous remarquerez que **la seule hypothèse** faite sur `ligne` est qu'elle accepte un argument nommé `couleur`.

```
In []: def ligne_rouge(*arguments, **keywords):
           # c'est le seul endroit où on fait une hypothèse sur la fonction 'ligne'
           keywords['couleur'] = "rouge"
           return ligne(*arguments, **keywords)
```

Ce qui permet maintenant de faire

```
In []: ligne_rouge(0, 100, 100, 0, epaisseur=4)
```

1.2.2 Pour en savoir plus - la forme générale

Une fois assimilé ce qui précède, vous avez de quoi comprendre une énorme majorité (99% au moins) du code python.

Dans le cas général, il est possible de combiner les 4 formes d'arguments: * des expressions (cas des arguments "normaux", dits positionnels) * des expressions nommées, comme `nom=<expression>` * des `*expressions` * des `**expressions`

Vous pouvez vous reporter à cette page pour une description détaillée de ce cas général.

À l'appel d'une fonction, il faut résoudre les arguments, c'est-à-dire associer une valeur à chaque paramètre formel (ceux qui apparaissent dans le `def`) à partir des valeurs figurant dans l'appel.

L'idée est que pour faire cela, les arguments de l'appel ne sont pas pris dans l'ordre où ils apparaissent, mais les arguments positionnels sont utilisés en premier. La logique est que, naturellement les arguments positionnnels (ou ceux qui proviennent d'une `*expression`) viennent sans nom, et donc ne peuvent pas être utilisés pour résoudre des arguments nommés.

Voici un tout petit exemple pour vous donner une idée de la complexité de ce mécanisme lorsqu'on mélange toutes les 4 formes d'arguments à l'appel de la fonction (alors qu'on a défini la fonction avec 4 paramètres positionnels)

```
In []: def foo(a, b, c, d):
           print(a, b, c, d)

In []: foo(1, c=3, *(2,), **{'d':4})

In []: foo (1, b=3, *(2,), **{'d':4})
```

Si le problème ne vous semble pas clair, vous pouvez regarder la documentation python décrivant ce problème.