W7-S6-C1-Complement-python3-vs-python2

December 15, 2014

1 python3 vs python2

1.1 Complément - niveau intermédiaire

Comme promis en Semaine 1, et maintenant que vous avez une vision d'ensemble de python2, voici un complément consacré à python3. Nous aborderons cette question selon deux angles : * pour commencer nous ferons un résumé des différences entre les deux langages, * et nous tenterons de faire le point sur la migration entre les deux versions.

Dans l'introduction de la Semaine 1, nous avons indiqué avoir choisi de concentrer le cours sur python2 car c'est encore aujourd'hui la version dominante du langage. Vous pourrez lire ici les résultats d'un sondage fait fin 2013/début 2014 qui le démontre assez nettement, même si les choses semblent bouger comme on le verra plus bas.

1.1.1 Les différences

Pendant toute la maturation de python, au moins depuis l'introduction de python2 en 2000, toutes les évolutions ont été faites avec compatibilité ascendante, et vous pouvez en théorie faire tourner du code 2.1 dans un interprèteur 2.7. Avec cette approche, il n'est naturellement pas possible d'enlever ou de changer les traits du langage qui ont été ratés :)

La décision de créer python3 a été prise dans le but de corriger ce genre de défauts, avec en contrepartie la nécessité de migrer tout la base de code. Commençons par faire un survol des changements, avant de voir dans la deuxième partie comment se passe cette migration.

Les différences sont résumées dans cette liste exhaustive, en voici une version abrégée.

`print` On a déjà eu l'occasion de l'évoquer, la différence la plus visible entre les deux versions du langage est que * dans python2 `print` est une **instruction** et ne prend pas de parenthèse, * alors qu'en python3 c'est une **fonction** et donc requiert des parenthèses.

Comme on l'avait signalé en Semaine 1 (Séquence "Pourquoi python ?"), vous avez la possibilité d'écrire du code python2 qui utilise `print` avec la syntaxe de python3 en mentionnant

```
from __future__ import print_function
```

Cela dit, `print` est une construction très visible pendant la période d'apprentissage, mais dans du vrai code son usage est **beaucoup moins répandu** qu'on ne pourrait le penser, on utilise la plupart du temps des modules de `logging` ou autres fonctions d'écriture sur fichier, aussi l'évolution de `print` est en réalité, entre python2 et python3, beaucoup moins cruciale qu'il n'y paraît.

types `str` et `unicode` Le changement le plus radical, dans le sens le changement auquel il est le plus délicat de s'adapter, est sans doute celui qui concerne la représentation des caractères.

Comme il est expliqué ici Python3 distingue entre les concepts de *text* et de *(binary) data*, là où python2 distinguait entre les types `str` et `unicode`. La phrase d'introduction vous donne une idée de l'étendue des changements qui ont été faits dans ce domaine :

C'est d'ailleurs une des raisons pour lesquelles dans ce MOOC nous avons choisi de soigneusement laisser de coté le type `unicode`.

Dans le compléments sur les accents en Semaine 1 (Séquence "Les outils de la distribution standard python"), nous avions expliqué que le modèle mental, selon lequel un caractère est équivalent à un octet est, avec les encodages modernes Unicode, devenu obsolète. C'est en fait ce modèle mental qui était véhiculé par le type `str` de python2.

En python3, le type `unicode` n'existe plus (enfin il serait plus précis de dire que le type `str` n'existe plus et que le type `unicode` de python2 s'appelle `str` en python3); et on a introduit le nouveau type `bytes` qui permet de manipuler de la donnée binaire pure. Nous vous renvoyons à l'article cité ci-dessus pour davantage de détails sur cet aspect de la migration.

types `int` et `long` Le type `long` a disparu (enfin, un peu comme avec les chaînes, `long` a remplacé `int` et on a supprimé le nom `long`). Cette différence a généralement assez peu d'impact dans un portage.

classes *new-style* On l'a déjà mentionné dans le cours, toutes les classes en python3 sont des **classes *new-style***, qu'elles héritent ou non de `object`; on n'a donc plus besoin d'hériter d'object, mais on peut le faire si on récupère du vieux code.

Là encore l'impact de ce changement lors d'un portage est généralament faible ou nul.

Utilisation massive des itérateurs Un grand nombre de fonctions et méthodes - qu'elles soient *builtin* ou qu'elle viennent d'une librairie - retournent en python2 des listes. C'est le cas, on l'a vu, pour * `range()`, * `dict.items()` ou ses voisines `values()` et `keys()`, et un très grand nombre d'autres.

En python3, dans tous les cas où c'était possible, on a préféré **retourner des itérateurs** en lieu et place des listes.

Ce qui a du même coup permis de supprimer les fonctions et méthodes python2 qui avaient été rajoutées après coup, pour proposer ce mode de fonctionnement sans casser la compatibilité. Pour prendre des exemples, en python2 vous aviez le choix entre

```
In []: range(10)
```

et

```
In []: xrange(10)
```

parce que `xrange` avait été introduit **après** les itérateurs. En python3, il n'y a plus que `range`, qui renvoie un itérateur; il n'y a plus non plus, par exemple, `dict.itervalues` mais seulement `dict.values` qui renvoie un itérateur.

Variables locales à une compréhension Pour des raisons principalement historiques, les variables de boucle `fuitent` (*leak* en anglais), c'est-à-dire restaient définies à la sortie d'une boucle, comme on l'a vu en Semaine 3 (séquence "Les boucles `for` et les itérateurs");

Ceci est modifié en python3 pour les compréhensions, comme on le voit sur cet exemple :

python2

python3

On observe bien entendu le même comportement avec les compréhensions de dictionnaires ou d'ensembles. Par contre, on aurait pu penser que le même comportement serait adopté pour les boucles `for` à part entière, ce n'est **pas le cas malheureusement** :

python2

python3

Autres changements Citons également, en vrac : * les orthographes ont été homogénéisées, comme le fait qu'un itérateur doit maintenant définir `__next__()` et non `next()`; * on peut utiliser des caractères Unicode dans les noms de variables, donc des accents, des cédilles, on peut appeler une fonction Σ - non pas que cela semble forcément une très bonne idée à première vue mais bon; * en python3, on a la possibilité d'annoter, au niveau syntaxique les arguments et valeur de retour des fonctions (voir PEP3107); dans l'état actuel il s'agit d'annotations à vocation **surtout documentaire** et il ne semble pas que le langage se dirige

vers un contrôle de type plus strict dans ce domaine; * une nouvelle notation a été introduire pour spécifier la métaclasse; * et tout un tas d'autres améliorations moins significatives dont vous trouverez une liste plus exhaustive ici (c'est la même référence que celle donnée ci-dessus dans le chapeau sur "Les différences".

Conclusion Comme vous le voyez, ce que vous avez appris dans ce MOOC est pratiquement utilisable tel quel dans un environnement python3. Si vous devez coder en python3 à partir des connaissances acquises dans ce cours, il vous reste principalement à maîtriser les deux types `str` et `bytes`, et les notions d'encodage qui y sont rattachées.

1.1.2 Un point sur la migration

Les gros joueurs Jusqu'à récemment, quand on commençait un nouveau projet, on tombait rapidement sur une librairie majeure dont on avait besoin et qui n'était pas disponible en python3; du coup, même des projets relativement récents ont choisi de cibler python2.

Il me semble que la situation est en train de changer un peu; en tous cas, les deux gros joueurs que sont `django`, *the web framework for perfectionists with deadlines*, et `NumPy` - *fundamental package for scientific computing with Python* , sont à présent disponibles en python3 :

- `django` depuis la version 1.5
- `numpy` depuis 1.5.0 et scipy depuis 0.9.0

Cela dit, la date de *End-Of-Life* pour python2.7 qui représente la date de fin de maintenance qui avait été initialement fixée à 2015 lors de la sortie de python3, a été dans un update du PEP373 en date d'Avril 2014 retardée de cinq ans pour courir jusqu'en 2020.

Outils et stratégies Au départ, l'idée étant de nettoyer le langage, on a incité les gens à **basculer** d'une version à l'autre; il existe pour cela un outil qui s'appelle `2to3` - qui fait partie de la distribution standard - et qui permet de **traduire** un code python2 en python3 en le **modifiant**. `2to3` ne sait pas forcément résoudre tous les problèmes mais permet d'en évacuer la plus grosse partie.

Le souci toutefois avec cette approche est qu'une fois qu'on a fait la traduction, on se retrouve avec **deux codes**. Pour tous les projets qui sont **réutilisables** - qui offrent des librairies, soit parce que c'est l'objectif principal du projet, soit comme *by-product* - cette approche signifie de **maintenir** ces deux codes, ce qui est le plus souvent inacceptable.

Aussi, ces derniers temps, il semble qu'on se dirige plus vers une approche de **code unique**, au travers de librairies comme `six` - disponible via `pypi`.

1.1.3 Pour en savoir plus

Vous pouvez consulter également : * Should I use Python 2 or Python 3 for my development activity? dans le wiki python ; * Le sentiment du BDFL Guido Van Rossum dans son talk à PyCon 2014, aux environs de 09:00 ; * Un guide de portage de python2 à python3.