Portage de code Python 2 vers Python 3

Version 3.10.16

Guido van Rossum
and the Python development team

décembre 07, 2024

Python Software Foundation
Email : docs@python.org

Table des matières

auteur Brett Cannon

Résumé

Python 3 étant le futur de Python tandis que Python 2 est encore activement utilisé, il est préférable de faire en sorte que votre projet soit disponible pour les deux versions majeures de Python. Ce guide est destiné à vous aider à comprendre comment gérer simultanément Python 2 & 3.

Si vous cherchez à porter un module d'extension plutôt que du pur Python, veuillez consulter cporting-howto.

Si vous souhaitez lire l'avis d'un développeur principal de Python sur ce qui a motivé la création de Python 3, vous pouvez lire le Python 3 Q & A de Nick Coghlan ou bien Why Python 3 exists de Brett Cannon.

Vous pouvez lire les archives de la liste diffusion python-porting dans vos recherches sur les questions liées au portage.

1 La version courte

Afin de rendre votre projet compatible Python 2/3 avec le même code source, les étapes de base sont :

1. Ne se préoccuper que du support de Python 2.7
2. S'assurer d'une bonne couverture des tests (coverage.py peut aider ; `python -m pip install coverage`)
3. Apprendre les différences entre Python 2 et 3
4. Utiliser Futurize (ou Modernize) pour mettre à jour votre code (par exemple `python -m pip install future`)
5. Utilisez Pylint pour vous assurer que vous ne régressez pas sur votre prise en charge de Python 3 (`python -m pip install pylint`)
6. Utiliser caniusepython3 pour déterminer quelles sont, parmi les dépendances que vous utilisez, celles qui bloquent votre utilisation de Python 3 (`python -m pip install caniusepython3`)
7. Une fois que vos dépendances ne sont plus un obstacle, utiliser l'intégration continue pour s'assurer que votre code demeure compatible Python 2 & 3 (tox peut aider à tester la comptabilité de sources avec plusieurs versions de Python ; `python -m pip install tox`)
8. Envisager l'utilisation d'un vérifieur de type statique afin de vous assurer que votre façon d'utiliser les types est compatible avec Python 2 et 3 (par exemple en utilisant mypy pour vérifier votre typage sous Python 2 et 3 ; `python -m pip install mypy`).

Note : Note : L'utilisation de `python -m pip install` garantit que le `pip` invoqué est bien celui installé avec la version de Python que vous utilisez, que ce soit un `pip` du système ou un `pip` installé dans un environnement virtuel.

---

2 Détails

Un point clé du support simultané de Python 2 et 3 est qu'il vous est possible de commencer **dès aujourd'hui** ! Même si vos dépendances ne sont pas encore compatibles Python 3, vous pouvez moderniser votre code **dès maintenant** pour gérer Python 3. La plupart des modifications nécessaires à la compatibilité Python 3 donnent un code plus propre utilisant une syntaxe plus récente, même dans du code Python 2.

Un autre point important est que la modernisation de votre code Python 2 pour le rendre compatible Python 3 est pratiquement automatique. Bien qu'il soit possible d'avoir à effectuer des changements d'API compte-tenu de la clarification de la gestion des données textuelles et binaires dans Python 3, le travail de bas niveau est en grande partie fait pour vous et vous pouvez ainsi bénéficiez de ces modifications automatiques immédiatement.

Gardez ces points clés en tête pendant que vous lisez les détails ci-dessous concernant le portage de votre code vers une compatibilité simultanée Python 2 et 3.

2.1 Abandon de la compatibilité Python 2.6 et antérieures

Bien qu'il soit possible de rendre Python 2.5 compatible avec Python 3, il est **beaucoup** plus simple de n'avoir qu'à travailler avec Python 2.7. Si abandonner Python 2.5 n'est pas une option, alors le projet six peut vous aider à gérer simultanément Python 2.5 et 3 (`python -m pip install six`). Néanmoins, soyez conscient que la quasi-totalité des projets listés dans ce guide pratique ne seront pas applicables à votre situation.

Si vous pouvez ignorer Python 2.5 et antérieur, les changements nécessaires à appliquer à votre code devraient encore ressembler à vos yeux à du code Python idiomatique. Dans le pire cas, vous devrez utiliser une fonction plutôt qu'une méthode dans certains cas, ou bien vous devrez importer une fonction plutôt qu'utiliser une fonction native, mais le reste du temps le code transformé devrait vous rester familier.

Mais nous vous conseillons de viser seulement un support de Python 2.7. Python 2.6 n'est plus supporté gratuitement et par conséquent ne reçoit plus aucun correctif. Cela signifie que **vous** devrez trouver des solutions de contournement aux problèmes que vous rencontrez avec Python 2.6. Il existe en outre des outils mentionnés dans ce guide pratique qui

ne supportent pas Python 2.6 (par exemple Pylint), ce qui sera de plus en plus courant au fil du temps. Il est simplement plus facile pour vous de n'assurer une compatibilité qu'avec les versions de Python que vous avez l'obligation de gérer.

2.2 Assurez vous de spécifier la bonne version supportée dans le fichier `setup.py`

Votre fichier `setup.py` devrait contenir le bon trove classifier spécifiant les versions de Python avec lesquelles vous êtes compatible. Comme votre projet ne supporte pas encore Python 3, vous devriez au moins spécifier `Programming Language :: Python :: 2 :: Only`. Dans l'idéal vous devriez indiquer chaque version majeure/mineure de Python que vous gérez, par exemple `Programming Language :: Python :: 2.7`.

2.3 Obtenir une bonne couverture de code

Une fois que votre code est compatible avec la plus ancienne version de Python 2 que vous souhaitez, vous devez vous assurer que votre suite de test a une couverture suffisante. Une bonne règle empirique consiste à avoir suffisamment confiance en la suite de test pour qu'une erreur apparaissant après la réécriture du code par les outils automatiques résulte de bogues de ces derniers et non de votre code. Si vous souhaitez une valeur cible, essayez de dépasser les 80 % de couverture (et ne vous sentez pas coupable si vous trouvez difficile de faire mieux que 90 % de couverture). Si vous ne disposez pas encore d'un outil pour mesurer la couverture de code, coverage.py est recommandé.

2.4 Apprendre les différences entre Python 2 et 3

Une fois que votre code est bien testé, vous êtes prêt à démarrer votre portage vers Python 3 ! Mais afin de comprendre comment votre code va changer et à quoi s'intéresser spécifiquement pendant que vous codez, vous aurez sûrement envie de découvrir quels sont les changements introduits par Python 3 par rapport à Python 2. Pour atteindre cet objectif, les deux meilleurs moyens sont de lire le document whatsnew-index de chaque version de Python 3 et le livre Porting to Python 3 (gratuit en ligne, en anglais). Il y a également une « antisèche » (cheat sheet, ressource en anglais) très pratique du projet Python-Future.

2.5 Mettre à jour votre code

Une fois que vous pensez en savoir suffisamment sur les différences entre Python 3 et Python 2, il est temps de mettre à jour votre code ! Vous avez le choix entre deux outils pour porter votre code automatiquement : Futurize et Modernize. Le choix de l'outil dépend de la dose de Python 3 que vous souhaitez introduire dans votre code. Futurize s'efforce d'introduire les idiomes et pratiques de Python 3 dans Python 2, par exemple en réintroduisant le type `bytes` de Python 3 de telle sorte que la sémantique soit identique entre les deux versions majeures de Python. En revanche, Modernize est plus conservateur et vise un sous-ensemble d'instructions Python 2/3, en s'appuyant directement sur six pour la compatibilité. Python 3 étant le futur de Python, il pourrait être préférable d'utiliser *Futurize* afin de commencer à s'ajuster aux nouvelles pratiques introduites par Python 3 avec lesquelles vous n'êtes pas encore habitué.

Indépendamment de l'outil sur lequel se porte votre choix, celui-ci mettra à jour votre code afin qu'il puisse être exécuté par Python 3 tout en maintenant sa compatibilité avec la version de Python 2 dont vous êtes parti. En fonction du niveau de prudence que vous visez, vous pouvez exécuter l'outil sur votre suite de test d'abord puis inspecter visuellement la différence afin de vous assurer que la transformation est exacte. Après avoir transformé votre suite de test et vérifié que tous les tests s'exécutent comme attendu, vous pouvez transformer le code de votre application avec l'assurance que chaque test qui échoue correspond à un échec de traduction.

Malheureusement les outils ne peuvent pas automatiser tous les changements requis pour permettre à votre code de s'exécuter sous Python 3 et il y a donc quelques points sur lesquels vous devrez travailler manuellement afin d'atteindre la compatibilité totale Python 3 (les étapes nécessaires peuvent varier en fonction de l'outil utilisé). Lisez la documentation de l'outil que vous avez choisi afin d'identifier ce qu'il corrige par défaut et ce qui peut être appliqué de façon optionnelle afin de savoir ce qui sera (ou non) corrigé pour vous ou ce que vous devrez modifier vous-même (par exemple, le remplacement `io.open()` plutôt que la fonction native `open()` est inactif par défaut dans

Modernize). Heureusement, il n'y a que quelques points à surveiller qui peuvent réellement être considérés comme des problèmes difficiles à déboguer si vous n'y prêtez pas attention.

Division

Dans Python 3, `5 / 2 == 2.5` et non `2` ; toutes les divisions entre des valeurs `int` renvoient un `float`. Ce changement était en réalité planifié depuis Python 2.2, publié en 2002. Depuis cette date, les utilisateurs ont été encouragés à ajouter `from __future__ import division` à tous les fichiers utilisant les opérateurs `/` et `//` ou à exécuter l'interpréteur avec l'option `-Q`. Si vous n'avez pas suivi cette recommandation, vous devrez manuellement modifier votre code et effectuer deux changements :

1. Ajouter `from __future__ import division` à vos fichiers
2. Remplacer tous les opérateurs de division par `//` pour la division entière, le cas échéant, ou utiliser `/` et vous attendre à un résultat flottant

La raison pour laquelle `/` n'est pas simplement remplacé par `//` automatiquement est que si un objet définit une méthode `__truediv__` mais pas de méthode `__floordiv__`, alors votre code pourrait produire une erreur (par exemple, une classe définie par l'utilisateur qui utilise `/` pour définir une opération quelconque mais pour laquelle `//` n'a pas du tout la même signification, voire n'est pas utilisé du tout).

Texte et données binaires

Dans Python 2, il était possible d'utiliser le type `str` pour du texte et pour des données binaires. Malheureusement cet amalgame entre deux concepts différents peut conduire à du code fragile pouvant parfois fonctionner pour les deux types de données et parfois non. Cela a également conduit à des API confuses si les auteurs ne déclaraient pas explicitement que quelque chose qui acceptait `str` était compatible avec du texte ou des données binaires et pas un seul des deux types. Cela a compliqué la situation pour les personnes devant gérer plusieurs langages avec des API qui ne se préoccupaient pas de la gestion de `unicode` lorsqu'elles affirmaient être compatibles avec des données au format texte.

Afin de rendre la distinction entre texte et données binaires claire et prononcée, Python 3 a suivi la voie pavée par la plupart des langages créés à l'ère d'Internet et a séparé les types texte et données binaires de telle sorte qu'il ne soit plus possible de les confondre (Python est antérieur à la démocratisation de l'accès à Internet). Cette séparation ne pose pas de problème pour du code ne gérant soit que du texte, soit que des données binaires. Cependant un code source devant gérer les deux doit désormais se préoccuper du type des données manipulées, ce qui explique que ce processus ne peut pas être entièrement automatisé.

Pour commencer, vous devrez choisir quelles API travaillent sur du texte et lesquelles travaillent avec des données binaires (il est **fortement** recommandé de ne pas concevoir d'API qui gèrent les deux types compte-tenu de la difficulté supplémentaire que cela induit). Dans Python 2, cela signifie s'assurer que les API recevant du texte en entrée peuvent gérer `unicode` et celles qui reçoivent des données binaires fonctionnent avec le type `bytes` de Python 3 (qui est un sous-ensemble de `str` dans Python 2 et opère comme un alias du type `bytes` de Python 2). En général, le principal problème consiste à inventorier quelles méthodes existent et opèrent sur quel type dans Python & 3 simultanément (pour le texte, il s'agit de `unicode` dans Python 2 et `str` dans Python 3, pour le binaire il s'agit de `str`/`bytes` dans Python 2 et `bytes` dans Python 3). Le tableau ci-dessous liste les méthodes **spécifiques** à chaque type de données dans Python 2 et 3 (par exemple, la méthode `decode()` peut être utilisée sur des données binaires équivalentes en Python 2 et 3, mais ne peut pas être utilisée de la même façon sur le type texte en Python 2 et 3 car le type `str` de Python 3 ne possède pas de telle méthode). Notez que depuis Python 3.5, la méthode `__mod__` a été ajoutée au type *bytes*.

Format texte	Format binaire
	decode
encode	
format	
isdecimal	
isnumeric	

Vous pouvez rendre le problème plus simple à gérer en réalisant les opérations d'encodage et de décodage entre données binaires et texte aux extrémités de votre code. Cela signifie que lorsque vous recevez du texte dans un format

binaire, vous devez immédiatement le décoder. À l'inverse si votre code doit transmettre du texte sous forme binaire, encodez-le le plus tard possible. Cela vous permet de ne manipuler que du texte à l'intérieur de votre code et permet de ne pas se préoccuper du type des données sur lesquelles vous travaillez.

Le point suivant est de s'assurer que vous savez quelles chaînes de caractères littérales de votre code correspondent à du texte ou à du binaire. Vous devez préfixer par `b` tous les littéraux qui représentent des données binaires et par `u` les littéraux qui représentent du texte (il existe une importation du module `__future__` permettant de forcer l'encodage de toutes les chaînes de caractères littérales non spécifiées en Unicode, mais cette pratique s'est avérée moins efficace que l'ajout explicite des préfixe `b` et `u`).

Une conséquence de cette dichotomie est que vous devez être prudents lors de l'ouverture d'un fichier. À moins que vous travailliez sous Windows, il y a des chances pour que vous ne vous soyez jamais préoccupé de spécifier le mode `b` lorsque vous ouvrez des fichiers binaires (par exemple `rb` pour lire un fichier binaire). Sous Python 3, les fichiers binaire et texte sont distincts et mutuellement incompatibles ; se référer au module `io` pour plus de détails. Ainsi vous **devez** décider lorsque vous ouvrez un fichier si vous y accéderez en mode binaire (ce qui permet de lire et écrire des données binaires) ou en mode texte (ce qui permet de lire et écrire du texte). Vous devez également utiliser `io.open()` pour ouvrir des fichiers plutôt que la fonction native `open()` étant donné que le module `io` est cohérent de Python 2 à 3, ce qui n'est pas vrai pour la fonction `open()` (en Python 3, il s'agit en réalité de `io.open()`). Ne cherchez pas à appliquer l'ancienne pratique consistant à utiliser `codecs.open()` qui n'est nécessaire que pour préserver une compatibilité avec Python 2.5.

Les constructeurs des types `str` et `bytes` possèdent une sémantique différente pour les mêmes arguments sous Python 2 et 3. Passer un entier à `bytes` sous Python 2 produit une représentation de cet entier en chaîne de caractères : `bytes(3) == '3'`. Mais sous Python 3, fournir un argument entier à `bytes` produit un objet *bytes* de la longueur de l'entier spécifié, rempli par des octets nuls : `bytes(3) == b'\x00\x00\x00'`. La même prudence est nécessaire lorsque vous passez un objet *bytes* à `str`. En Python 2, vous récupérez simplement l'objet *bytes* initial : `str(b'3') == b'3'`. Mais en Python 3, vous récupérez la représentation en chaîne de caractères de l'objet *bytes* : `str(b'3') == "b'3'"`.

Enfin, l'indiçage des données binaires exige une manipulation prudente (bien que le découpage, ou *slicing* en anglais, ne nécessite pas d'attention particulière). En Python 2, `b'123'[1] == b'2'` tandis qu'en Python 3 `b'123'[1] == 50`. Puisque les données binaires ne sont simplement qu'une collection de nombres en binaire, Python 3 renvoie la valeur entière de l'octet indicé. Mais en Python 2, étant donné que `bytes == str`, l'indiçage renvoie une tranche de longueur 1 de *bytes*. Le projet six dispose d'une fonction appelée `six.indexbytes()` qui renvoie un entier comme en Python 3 : `six.indexbytes(b'123', 1)`.

Pour résumer :

1. Décidez lesquelles de vos API travaillent sur du texte et lesquelles travaillent sur des données binaires
2. Assurez vous que votre code travaillant sur du texte fonctionne aussi avec le type `unicode` et que le code travaillant sur du binaire fonctionne avec le type `bytes` en Python 2 (voir le tableau ci-dessus pour la liste des méthodes utilisables par chaque type)
3. Préfixez tous vos littéraux binaires par `b` et toutes vos chaînes de caractères littérales par `u`
4. Décodez les données binaires en texte dès que possible, encodez votre texte au format binaire le plus tard possible
5. Ouvrez les fichiers avec la fonction `io.open()` et assurez-vous de spécifier le mode `b` le cas échéant
6. Utilisez avec prudence l'indiçage sur des données binaires

Utilisez la détection de fonctionnalités plutôt que la détection de version

Vous rencontrerez inévitablement du code devant décider quoi faire en fonction de la version de Python qui s'exécute. La meilleure façon de gérer ce cas est de détecter si les fonctionnalités dont vous avez besoin sont gérées par la version de Python sous laquelle le code s'exécute. Si pour certaines raisons cela ne fonctionne pas, alors vous devez tester si votre version est Python 2 et non Python 3. Afin de clarifier cette pratique, voici un exemple.

Supposons que vous avez besoin d'accéder à une fonctionnalité de `importlib` qui est disponible dans la bibliothèque standard de Python depuis la version 3.3, dans celle de Python 2 via le module importlib2 sur *PyPI*. Vous pourriez être tenté d'écrire un code qui accède, par exemple, au module `importlib.abc` avec l'approche suivante :

```
import sys

if sys.version_info[0] == 3:
    from importlib import abc
else:
    from importlib2 import abc
```

Le problème est le suivant : que se passe-t-il lorsque Python 4 est publié ? Il serait préférable de traiter le cas Python 2 comme l'exception plutôt que Python 3 et de supposer que les versions futures de Python 2 seront plus compatibles avec Python 3 qu'avec Python 2 :

```
import sys

if sys.version_info[0] > 2:
    from importlib import abc
else:
    from importlib2 import abc
```

Néanmoins la meilleure solution est de ne pas chercher à déterminer la version de Python mais plutôt à détecter les fonctionnalités disponibles. Cela évite les problèmes potentiels liés aux erreurs de détection de version et facilite la compatibilité future :

```
try:
    from importlib import abc
except ImportError:
    from importlib2 import abc
```

2.6 Prévenir les régressions de compatibilité

Une fois votre code traduit pour être compatible avec Python 3, vous devez vous assurer que votre code n'a pas régressé ou qu'il ne fonctionne pas sous Python 3. Ceci est particulièrement important si une de vos dépendances vous empêche de réellement exécuter le code sous Python 3 pour le moment.

Afin de vous aider à maintenir la compatibilité, nous préconisons que tous les nouveaux modules que vous créez aient au moins le bloc de code suivant en en-tête :

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
```

Vous pouvez également lancer Python 2 avec le paramètre `-3` afin d'être alerté en cas de divers problèmes de compatibilité que votre code déclenche durant son exécution. Si vous transformez les avertissements en erreur avec `-Werror`, vous pouvez être certain que ne passez pas accidentellement à côté d'un avertissement.

Vous pouvez également utiliser le projet Pylint et son option `--py3k` afin de modifier votre code pour recevoir des avertissements lorsque celui-ci dévie de la compatibilité Python 3. Cela vous évite par ailleurs d'appliquer Modernize ou Futurize sur votre code régulièrement pour détecter des régressions liées à la compatibilité. Cependant cela nécessite de votre part le support de Python 2.7 et Python 3.4 ou ultérieur étant donné qu'il s'agit de la version minimale gérée par Pylint.

2.7 Vérifier quelles dépendances empêchent la migration

Après avoir rendu votre code compatible avec Python 3, vous devez commencer à vous intéresser au portage de vos dépendances. Le projet caniusepython3 a été créé afin de vous aider à déterminer quels projets sont bloquants dans votre support de Python 3, directement ou indirectement. Il existe un outil en ligne de commande ainsi qu'une interface web : https://caniusepython3.com.

Le projet fournit également du code intégrable dans votre suite de test qui déclenchera un échec de test lorsque plus aucune de vos dépendances n'est bloquante pour l'utilisation de Python 3. Cela vous permet de ne pas avoir à vérifier manuellement vos dépendances et d'être notifié rapidement quand vous pouvez exécuter votre application avec Python 3.

2.8 Mettre à jour votre fichier `setup.py` pour spécifier la compatibilité avec Python 3

Une fois que votre code fonctionne sous Python 3, vous devez mettre à jour vos classeurs dans votre `setup.py` pour inclure `Programming Language :: Python :: 3` et non seulement le support de Python 2. Cela signifiera à quiconque utilise votre code que vous gérez Python 2 **et** 3. Dans l'idéal vous devrez aussi ajouter une mention pour chaque version majeure/mineure de Python que vous supportez désormais.

2.9 Utiliser l'intégration continue pour maintenir la compatibilité

Une fois que vous êtes en mesure d'exécuter votre code sous Python 3, vous devrez vous assurer que celui-ci fonctionne toujours pour Python 2 & 3. tox est vraisemblablement le meilleur outil pour exécuter vos tests avec plusieurs interpréteurs Python. Vous pouvez alors intégrer *tox* à votre système d'intégration continue afin de ne jamais accidentellement casser votre gestion de Python 2 ou 3.

Vous pouvez également utiliser l'option `-bb` de l'interpréteur Python 3 afin de déclencher une exception lorsque vous comparez des *bytes* à des chaînes de caractères ou à un entier (cette deuxième possibilité est disponible à partir de Python 3.5). Par défaut, des comparaisons entre types différents renvoient simplement `False` mais si vous avez fait une erreur dans votre séparation de la gestion texte/données binaires ou votre indiçage des *bytes*, vous ne trouverez pas facilement le bogue. Ce drapeau lève une exception lorsque ce genre de comparaison apparaît, facilitant ainsi son identification et sa localisation.

Et c'est à peu près tout ! Une fois ceci fait, votre code source est compatible avec Python 2 et 3 simultanément. Votre suite de test est également en place de telle sorte que vous ne cassiez pas la compatibilité Python 2 ou 3 indépendamment de la version que vous utilisez pendant le développement.

2.10 Envisager l'utilisation d'un vérificateur de type statique optionnel

Une autre façon de faciliter le portage de votre code est d'utiliser un vérificateur de type statique comme mypy ou pytype. Ces outils peuvent être utilisés pour analyser votre code comme s'il était exécuté sous Python 2, puis une seconde fois comme s'il était exécuté sous Python 3. L'utilisation double d'un vérificateur de type statique de cette façon permet de détecter si, par exemple, vous faites une utilisation inappropriée des types de données binaires dans une version de Python par rapport à l'autre. Si vous ajoutez les indices optionnels de typage à votre code, vous pouvez alors explicitement déclarer que vos API attendent des données binaires ou du texte, ce qui facilite alors la vérification du comportement de votre code dans les deux versions de Python.