Portage de code Python 2 vers Python 3

Version 2.7.18

Guido van Rossum
and the Python development team

mai 20, 2020

Python Software Foundation
Email : docs@python.org

Table des matières

author 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.

If you would like to read one core Python developer's take on why Python 3 came into existence, you can read Nick Coghlan's Python 3 Q & A or Brett Cannon's Why Python 3 exists.

Vous pouvez solliciter par courriel l'aide de la liste de diffusion python-porting pour vos 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. Only worry about supporting Python 2.7
2. S'assurer d'une bonne couverture des tests (coverage.py peut aider ; `pip install coverage`)
3. Apprendre les différences entre Python 2 et 3
4. Use Futurize (or Modernize) to update your code (e.g. `pip install future`)
5. Use Pylint to help make sure you don't regress on your Python 3 support (`pip install pylint`)
6. Utiliser caniusepython3 pour déterminer quelles sont, parmi les dépendances que vous utilisez, celles qui bloquent votre utilisation de Python 3 (`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 ; `pip install tox`)
8. Consider using optional static type checking to make sure your type usage works in both Python 2 & 3 (e.g. use mypy to check your typing under both Python 2 & Python 3).

2 Détails

A key point about supporting Python 2 & 3 simultaneously is that you can start **today** ! Even if your dependencies are not supporting Python 3 yet that does not mean you can't modernize your code **now** to support Python 3. Most changes required to support Python 3 lead to cleaner code using newer practices even in Python 2 code.

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 Drop support for Python 2.6 and older

While you can make Python 2.5 work with Python 3, it is **much** easier if you only have to work with Python 2.7. If dropping Python 2.5 is not an option then the six project can help you support Python 2.5 & 3 simultaneously (`pip install six`). Do realize, though, that nearly all the projects listed in this HOWTO will not be available to you.

If you are able to skip Python 2.5 and older, then the required changes to your code should continue to look and feel like idiomatic Python code. At worst you will have to use a function instead of a method in some instances or have to import a function instead of using a built-in one, but otherwise the overall transformation should not feel foreign to you.

But you should aim for only supporting Python 2.7. Python 2.6 is no longer freely supported and thus is not receiving bugfixes. This means **you** will have to work around any issues you come across with Python 2.6. There are also some tools mentioned in this HOWTO which do not support Python 2.6 (e.g., Pylint), and this will become more commonplace as time goes on. It will simply be easier for you if you only support the versions of Python that you have to support.

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

Once you have your code supporting the oldest version of Python 2 you want it to, you will want to make sure your test suite has good coverage. A good rule of thumb is that if you want to be confident enough in your test suite that any failures that appear after having tools rewrite your code are actual bugs in the tools and not in your code. If you want a number to aim for, try to get over 80% coverage (and don't feel bad if you find it hard to get better than 90% coverage). If you don't already have a tool to measure test coverage then coverage.py is recommended.

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 « What's New » de chaque version de Python 3 et le livre Porting to Python 3 (gratuit en ligne). Il y a également une cheat sheet très pratique du projet Python-Future.

2.5 Mettre à jour votre code

Once you feel like you know what is different in Python 3 compared to Python 2, it's time to update your code ! You have a choice between two tools in porting your code automatically : Futurize and Modernize. Which tool you choose will depend on how much like Python 3 you want your code to be. Futurize does its best to make Python 3 idioms and practices exist in Python 2, e.g. backporting the `bytes` type from Python 3 so that you have semantic parity between the major versions of Python. Modernize, on the other hand, is more conservative and targets a Python 2/3 subset of Python, directly relying on six to help provide compatibility. As Python 3 is the future, it might be best to consider Futurize to begin adjusting to any new practices that Python 3 introduces which you are not accustomed to yet.

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 échant, ou utiliser `/` et vous attendre à un résultat flottant

The reason that `/` isn't simply translated to `//` automatically is that if an object defines a `__truediv__` method but not `__floordiv__` then your code would begin to fail (e.g. a user-defined class that uses `/` to signify some operation but not `//` for the same thing or at all).

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.

To make the distinction between text and binary data clearer and more pronounced, Python 3 did what most languages created in the age of the internet have done and made text and binary data distinct types that cannot blindly be mixed together (Python predates widespread access to the internet). For any code that deals only with text or only binary data, this separation doesn't pose an issue. But for code that has to deal with both, it does mean you might have to now care about when you are using text compared to binary data, which is why this cannot be entirely automated.

To start, you will need to decide which APIs take text and which take binary (it is **highly** recommended you don't design APIs that can take both due to the difficulty of keeping the code working; as stated earlier it is difficult to do well). In Python 2 this means making sure the APIs that take text can work with `unicode` and those that work with binary data work with the `bytes` type from Python 3 (which is a subset of `str` in Python 2 and acts as an alias for `bytes` type in Python 2). Usually the biggest issue is realizing which methods exist on which types in Python 2 & 3 simultaneously (for text that's `unicode` in Python 2 and `str` in Python 3, for binary that's `str`/`bytes` in Python 2 and `bytes` in Python 3). The following table lists the **unique** methods of each data type across Python 2 & 3 (e.g., the `decode()` method is usable on the equivalent binary data type in either Python 2 or 3, but it can't be used by the textual data type consistently between Python 2 and 3 because `str` in Python 3 doesn't have the method). Do note that as of Python 3.5 the `__mod__` method was added to the bytes type.

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.

The next issue is making sure you know whether the string literals in your code represent text or binary data. You should add a `b` prefix to any literal that presents binary data. For text you should add a `u` prefix to the text literal. (there is a `__future__` import to force all unspecified literals to be Unicode, but usage has shown it isn't as effective as adding a `b` or `u` prefix to all literals explicitly)

As part of this dichotomy you also need to be careful about opening files. Unless you have been working on Windows, there is a chance you have not always bothered to add the `b` mode when opening a binary file (e.g., `rb` for binary reading). Under Python 3, binary files and text files are clearly distinct and mutually incompatible; see the `io` module for details. Therefore, you **must** make a decision of whether a file will be used for binary access (allowing binary data to be read and/or written) or textual access (allowing text data to be read and/or written). You should also use `io.open()` for opening files instead of the built-in `open()` function as the `io` module is consistent from Python 2 to 3 while the built-in `open()` function is not (in Python 3 it's actually `io.open()`). Do not bother with the outdated practice of using `codecs.open()` as that's only necessary for keeping compatibility with 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. Mark all binary literals with a `b` prefix, textual literals with a `u` prefix
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. Be careful when indexing into binary data

Use feature detection instead of version detection

Inevitably you will have code that has to choose what to do based on what version of Python is running. The best way to do this is with feature detection of whether the version of Python you're running under supports what you need. If for some reason that doesn't work then you should make the version check be against Python 2 and not Python 3. To help explain this, let's look at an example.

Let's pretend that you need access to a feature of importlib that is available in Python's standard library since Python 3.3 and available for Python 2 through importlib2 on PyPI. You might be tempted to write code to access e.g. the `importlib.abc` module by doing the following :

```
import sys

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

The problem with this code is what happens when Python 4 comes out ? It would be better to treat Python 2 as the exceptional case instead of Python 3 and assume that future Python versions will be more compatible with Python 3 than Python 2 :

```
import sys

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

The best solution, though, is to do no version detection at all and instead rely on feature detection. That avoids any potential issues of getting the version detection wrong and helps keep you future-compatible :

```
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

After you have made your code compatible with Python 3 you should begin to care about whether your dependencies have also been ported. The caniusepython3 project was created to help you determine which projects – directly or indirectly – are blocking you from supporting Python 3. There is both a command-line tool as well as a web interface at 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.

You may also want to use the `-bb` flag with the Python 3 interpreter to trigger an exception when you are comparing bytes to strings or bytes to an int (the latter is available starting in Python 3.5). By default type-differing comparisons simply return `False`, but if you made a mistake in your separation of text/binary data handling or indexing on bytes you wouldn't easily find the mistake. This flag will raise an exception when these kinds of comparisons occur, making the mistake much easier to track down.

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 Consider using optional static type checking

Another way to help port your code is to use a static type checker like mypy or pytype on your code. These tools can be used to analyze your code as if it's being run under Python 2, then you can run the tool a second time as if your code is running under Python 3. By running a static type checker twice like this you can discover if you're e.g. misusing binary data type in one version of Python compared to another. If you add optional type hints to your code you can also explicitly state whether your APIs use textual or binary data, helping to make sure everything functions as expected in both versions of Python.