How to port Python 2 Code to Python 3

Version 3.12.0

Guido van Rossum and the Python development team

octobre 26, 2023

Python Software Foundation
Email : docs@python.org

Table des matières

auteur Brett Cannon

Résumé

Python 2 reached its official end-of-life at the start of 2020. This means that no new bug reports, fixes, or changes will be made to Python 2 - it's no longer supported.

This guide is intended to provide you with a path to Python 3 for your code, that includes compatibility with Python 2 as a first step.

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

The archived python-porting mailing list may contain some useful guidance.

1 La version courte

To achieve Python 2/3 compatibility in a single code base, the basic steps are :

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. Learn the differences between Python 2 and 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. Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 and 3 (tox can help test against multiple versions of Python ; `python -m pip install tox`)
8. Consider using optional static type checking to make sure your type usage works in both Python 2 and 3 (e.g. use mypy to check your typing under both Python 2 and Python 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

Even if other factors - say, dependencies over which you have no control - still require you to support Python 2, that does not prevent you taking the step of including Python 3 support.

Most changes required to support Python 3 lead to cleaner code using newer practices even in Python 2 code.

2.1 Different versions of Python 2

Ideally, your code should be compatible with Python 2.7, which was the last supported version of Python 2.

Some of the tools mentioned in this guide will not work with Python 2.6.

If absolutely necessary, the six project can help you support Python 2.5 and 3 simultaneously. Do realize, though, that nearly all the projects listed in this guide will not be available to you.

If you are able to skip Python 2.5 and older, the required changes to your code will be minimal. 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.

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 Be aware of the differences between Python 2 and 3

Once you have your code well-tested you are ready to begin porting your code to Python 3 ! But to fully understand how your code is going to change and what you want to look out for while you code, you will want to learn what changes Python 3 makes in terms of Python 2.

Some resources for understanding the differences and their implications for you code :

- the "What's New" doc for each release of Python 3
- the Porting to Python 3 book (which is free online)
- the handy cheat sheet from the Python-Future project.

2.5 Mettre à jour votre code

There are tools available that can port your code automatically.

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. This is the better approach for most cases.

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.

A good approach is to run the tool over your test suite first and visually inspect the diff to make sure the transformation is accurate. After you have transformed your test suite and verified that all the tests still pass as expected, then you can transform your application code knowing that any tests which fail is a translation failure.

Unfortunately the tools can't automate everything to make your code work under Python 3, and you will also need to read the tools' documentation in case some options you need are turned off by default.

Key issues to be aware of and check for :

Division

In Python 3, `5 / 2 == 2.5` and not `2` as it was in Python 2 ; all division between `int` values result in a `float`. This change has actually been planned since Python 2.2 which was released in 2002. Since then users have been encouraged to add `from __future__ import division` to any and all files which use the `/` and `//` operators or to be running the interpreter with the `-Q` flag. If you have not been doing this then you will need to go through your code and do two things :

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.

Python 3 made text and binary data distinct types that cannot simply be mixed together. 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.

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 and 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 and 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)

You also need to be careful about opening files. Possibly 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.

The constructors of both `str` and `bytes` have different semantics for the same arguments between Python 2 and 3. Passing an integer to `bytes` in Python 2 will give you the string representation of the integer : `bytes(3) == '3'`. But in Python 3, an integer argument to `bytes` will give you a bytes object as long as the integer specified, filled with null bytes : `bytes(3) == b'\x00\x00\x00'`. A similar worry is necessary when passing a bytes object to `str`. In Python 2 you just get the bytes object back : `str(b'3') == b'3'`. But in Python 3 you get the string representation of the bytes object : `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

Once you are able to fully run under Python 3 you will want to make sure your code always works under both Python 2 and 3. Probably the best tool for running your tests under multiple Python interpreters is tox. You can then integrate tox with your continuous integration system so that you never accidentally break Python 2 or 3 support.

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.

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.