# 如何将 Python 2 代码移植到 Python 3

**发布 *3.12.1***

**Guido van Rossum and the Python development team**

**十二月 24, 2023**

**Python Software Foundation**
**Email: docs@python.org**

## Contents



**作者** Brett Cannon

**摘要**

Python 2 生命期在 2020 年初正式结束。这意味着 Python 2 将不再接受新的错误报告、修复或更改。本指南旨在为你的代码提供一条通往 Python 3 的路径，其中包括作为第一步的如何与 Python 2 兼容。

如果您希望迁移扩展模块而不是纯 Python 代码，请参阅 cporting-howto。

已归档的 python-porting 邮件列表可能包含一些有用的指导。

# 1 简要说明

要在单一代码库中实现 Python 2/3 兼容性，基本步骤如下：

1. 只担心支持 Python 2.7 的问题
2. 确保你有良好的测试覆盖率（可以用 coverage.py；`python -m pip install coverage`）。
3. 了解 Python 2 和 Python 3 之间的区别
4. 使用 Futurize (或 Modernize) 来更新你的代码 (例如 `python -m pip install future`)。
5. 使用 Pylint 来帮助确保你在 Python 3 支持上不倒退 (`python -m pip install pylint`)
6. 使用 caniusepython3 来找出你的哪些依赖关系阻碍了你对 Python 3 的使用 (`python -m pip install caniusepython3`)
7. 一旦你的依赖关系不再阻碍你，请使用持续集成来确保 Python 2 与 3 的兼容 (tox 可以帮助针对多个 Python 版本进行测试; `python -m pip install tox`)
8. 考虑使用可选的 静态类型检查来确保你的类型用法在 Python 2 和 3 中都能正常工作 (例如使用 mypy 同时在 Python 2 和 Python 3 中检查你的类型标注; `python -m pip install mypy`)。

---

**备注：** 注意：使用 `python -m pip install` 来确保你发起调用的 `pip` 就是当前使用的 Python 所安装的那一个，无论它是系统级的 `pip` 还是安装在 虚拟环境中的。

---

# 2 详情

即使有其他因素——比如说，你无法控制的依赖关系——仍然要求你支持 Python 2，也不妨碍你着手包括对 Python 3 的支持。

支持 Python 3 所需的大多数更改都会使得代码更简洁，甚至在 Python 2 代码中也能应用新的实践。

## 2.1 Python 2 的不同版本

在理想情况下，你的代码应当兼容 Python 2.7，这是 Python 2 最后一个受支持的版本。

本指南中提到的某些工具将不适用于 Python 2.6。

如果有绝对的必要，six 项目可以帮助您同时支持 Python 2.5 和 3。但是，需要注意，本指南中列出的几乎所有项目你都将无法使用。

如果你能跳过 Python 2.5 或更早版本，那么对代码的修改将是极其微小的。在最坏的情况下你将必须使用函数来代替某些实例的方法或者必须导入函数而不是使用某个内置函数。

## 2.2 确保你在你的 `setup.py` 文件中指定适当的版本支持

在你的 `setup.py` 文件中，你应该有适当的 trove classifier 指定你支持哪些版本的 Python。由于你的项目还不支持 Python 3，你至少应该指定 `Programming Language :: Python :: 2 :: Only`。理想情况下，你还应该指定你所支持的 Python 的每个主要/次要版本，例如：`Programming Language :: Python :: 2.7`。.

## 2.3 良好的测试覆盖率

一旦你的代码支持了你希望的 Python 2 的最老版本，你将希望确保你的测试套件有良好的覆盖率。一个好的经验法则是，如果你想对你的测试套件有足够的信心，在让工具重写你的代码后出现的任何故障都是工具中的实际错误，而不是你的代码中的错误。如果你想要一个目标数字，试着获得超过 80% 的覆盖率（如果你发现很难获得好于 90% 的覆盖率，也不要感到遗憾）。如果你还没有一个测量测试覆盖率的工具，那么推荐使用 coverage.py。

## 2.4 了解 Python 2 和 Python 3 之间的区别

当你的代码经过充分的测试之后你就可以开始将代码移植到 Python 3 了！但是为了充分理解你的代码将如何变化以及在编写代码时需要注意什么，你需要学习 Python 3 对 Python 2 做了哪些改变。

一些有助于理解这些差异及其对代码影响的资源：

- Python 3 每个发布版的 “有什么新变化” 文档
- Porting to Python 3 电子书（免费在线版）
- 来自 Python-Future 项目的方便的 cheat sheet。

## 2.5 更新代码

有一些工具可以自动移植你的代码。

Futurize 尽其所能地让 Python 3 的惯用语法和实践在 Python 2 中存在，例如从 Python 3 反向移植 `bytes` 类型从而使 Python 的主要版本在语义上保持一致。在大多数情况下这是更好的做法。

而在另一方面，Modernize 更为保守并以 Python 2/3 的 Python 子集为目标，它直接依赖 six 来帮助提供兼容性。

一个很的做法是先在测试套件上运行工具然后可视化地检查差异以确保转换准确无误。在转换测试套件并验证所有测试都能按预期通过后，你就可以转换应用程序代码了，因为你知道任何测试失败都是是转译的错误。

不幸的是这些工具并不能自动化任何操作来使你的代码能在 Python 3 下运行，而且你还需要阅读工具的文档以防某些你所需要的选项被默认关闭。

需要注意和检查的关键问题：

### 除法

在 Python 3 中，`5 / 2 == 2.5` 而不是 Python 2 中 `2`；所有 `int` 值之间的除法都会得到一个 `float` 值。这个变化实际上从 2002 年发布的 Python 2.2 就已经计划好了。从那时起我们就鼓励用户在所有使用 `/` 和 `//` 运算符的文件中添加 `from __future__ import division`，或者附带 `-Q` 旗标运行解释器。如果你没有这样做，那么你需要检查你的代码并做两件事：

1. 添加 `from __future__ import division` 到你的文件。
2. 根据需要更新任何除法运算符，要么使用 `//` 来使用向下取整除法，要么继续使用 `/` 并得到一个浮点数

之所以没有简单地将 `/` 自动翻译成 `//`，是因为如果一个对象定义了一个 `__truediv__` 方法，但没有定义 `__floordiv__`，那么你的代码就会运行失败（例如，一个用户定义的类用 `/` 来表示一些操作，但没有用 `//` 来表示同样的事情或根本没有定义）。

### 文本与二进制数据

在 Python 2 中，你可以对文本和二进制数据都使用 `str``类型。不幸的是，这两个不同概念的融合可能会导致脆弱的代码，有时对任何一种数据都有效，有时则无效。如果人们没有明确说明某种接受``str``东西可以接受文本或二进制数据，而不是一种特定的类型，这也会导致 API 的混乱。这使情况变得复杂，特别是对于任何支持多种语言的人来说，因为 API 在声称支持文本数据时不会显式支持``unicode`。

Python 3 将文本和二进制数据区分为不同的类型而不能简单地混合在一起。对于任何只处理文本或二进制数据的代码来说，这种区分并不构成问题。但对于必须同时处理这两者的代码来说，这就意味着你现在可能必须关注何时使用文本而不是二进制数据，这也是为什么这无法完全自动化地操作。

决定哪些 API 接受文本而哪些接受二进制数据 (**强烈**建议不要设计同时接受这两种数据的 API 因为这很难保证代码正常工作；如前所述这一点是很难做好的)。在 Python 2 中这意味着要确保处理文本的 API 可以使用 `unicode` 而处理二进制数据的 API 可以使用来自 Python 3 的 `bytes` 类型 (它是 Python 2 中 `str` 的子集并被作为 Python 2 中 `bytes` 类型的别名)。通常最大的问题是要意识到哪些方法在 Python 2 和 Python 3 的哪些类型中同时存在 (对于文本来说在 Python 2 中是 `unicode` 而在 Python 3 中则是 `str`，对于二进制数据来说在 Python 2 中是 `str`/`bytes` 而在 Python 3 中则是 `bytes`)。

下表列出了每种数据类型在 Python 2 和 3 中的 **特有**方法（例如，`decode()` 方法在 Python 2 或 3 中都可用于等价的二进制数据类型，但文本数据类型无法在 Python 2 和 3 之间以一致的方式使用它因为 Python 3 中的 `str` 没有该方法）。请注意从 Python 3.5 起 bytes 类型增加了 `__mod__` 方法。

| 文本数据 | 二进制数据 |
|---|---|
| | decode |
| encode | |
| format | |
| isdecimal | |
| isnumeric | |

通过在你的代码边缘对二进制数据和文本进行编码和解码，可以使这种区分更容易处理。这意味着，当你收到二进制数据的文本时，你应该立即对其进行解码。而如果你的代码需要将文本作为二进制数据发送，那么就尽可能晚地对其进行编码。这使得你的代码在内部只与文本打交道，从而不必再去跟踪你所处理的数据类型。

下一个问题是确保你知道你的代码中的字符串字面值是代表文本还是二进制数据。你应当给任何代表二进制数据的字面值添加 `b` 前缀。对于文本则应当给文本字面值添加 `u` 前缀。(有一个 `__future__` 导入可以强制所有未指定前缀的字面值为 Unicode，但实际使用情况表明它并不像给所有字面值显式地添加 `b` 或 `u` 前缀那样有效)。

对于打开文件你也需要小心。在打开二进制文件时你可能并不总是会特意添加 `b` 模式 (例如，`rb` 表示二进制读取)。在 Python 3 中，二进制文件和文本文件是截然不同且互不兼容的；详情参见 `io` 模块。因此，你 **必须**决定一个文件要用于二进制访问 (允许读取和/或写入二进制数据) 还是文本访问 (允许读取和/或写入文本数据)。你还应当使用 `io.open()` 来打开文件而不是使用内置的 `open()` 函数因为 `io` 模块从 Python 2 到 3 是保持一致的而内置的 `open()` 函数并非如此 (在 Python 3 中它实际上是 `io.open()`)。请不要使用 `codecs.open()` 这种过时的做法因为它仅在保持与 Python 2.5 的兼容时才是必要的。

`str` 和 `bytes` 的构造器在 Python 2 和 3 中对相同的参数有不同的语义。在 Python 2 中，向 `bytes` 传入一个整数会得到该整数的字符串表示形式: `bytes(3) == '3'`。但在 Python 3 中，向 `bytes` 传入一个整数参数则会得到一个该整数所指定的长度的字节对象，并空字节填充: `bytes(3) == b'\x00\x00\x00'`。当向 `str` 传入字节串对象时也同样需要注意。在 Python 2 中你只是得到该字节串对象: `str(b'3') == b'3'`。但在 Python 3 中你将得到该字节串对象的字符串表示形式: `str(b'3') == "b'3'"`。

最后，二进制数据的索引需要仔细处理（切片 **不需要**任何特殊处理）。在 Python 2 中 `b'123'[1] == b'2'`，而在 Python 3 中 `b'123'[1] == 50`。因为二进制数据只是二进制数的集合，Python 3 会返回你索引的字节的整数值。但是在 Python 2 中，因为 `bytes == str`，索引会返回一个单项的字节片断。six 项目有一个名为 `six.indexbytes()` 的函数，它将像在 Python 3 中一样返回一个整数: `six.indexbytes(b'123', 1)`。

总结一下:

1. 决定你的 API 中哪些采用文本，哪些采用二进制数据
2. 确保你对文本工作的代码也能对 `unicode` 工作，对二进制数据的代码在 Python 2 中能对 `bytes` 工作（关于每种类型不能使用的方法，见上表）。
3. 用 `b` 前缀标记所有二进制字词，用 `u` 前缀标记文本字词
4. 尽快将二进制数据解码为文本，尽可能晚地将文本编码为二进制数据
5. 使用 `io.open()` 打开文件，并确保在适当时候指定 `b` 模式。
6. 在对二进制数据进行索引时要小心

#### 使用特征检测而不是版本检测

你不可避免地会有一些代码需要根据运行的 Python 版本来选择要做什么。做到这一点的最好方法是对你运行的 Python 版本是否支持你所需要的东西进行特征检测。如果由于某种原因这不起作用，那么你应该让版本检测针对 Python 2 而不是 Python 3。为了帮助解释这个问题，让我们看一个例子。

假设你需要访问 `importlib` 的一个功能，该功能自 Python 3.3 开始在 Python 的标准库中提供，并且通过 PyPI 上的 importlib2 提供给 Python 2。你可能会想写代码来访问例如 `importlib.abc` 模块，方法如下:

```
import sys

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

这段代码的问题是，当 Python 4 出来的时候会发生什么？最好是将 Python 2 作为例外情况，而不是 Python 3，并假设未来的 Python 版本与 Python 3 的兼容性比 Python 2 更强:

```
import sys

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

不过，最好的解决办法是根本不做版本检测，而是依靠特征检测。这就避免了任何潜在的版本检测错误的问题，并有助于保持你对未来的兼容:

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

### 2.6 防止兼容性退步

一旦你完全翻译了你的代码，使之与 Python 3 兼容，你将希望确保你的代码不会退步，不会在 Python 3 上停止工作。如果你有一个依赖关系阻碍了你目前在 Python 3 上的实际运行，那就更是如此了。

为了帮助保持兼容，你创建的任何新模块都应该在其顶部至少有以下代码块:

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

你也可以在运行 Python 2 时使用 `-3` 标志，对你的代码在执行过程中引发的各种兼容性问题进行警告。如果你用 `-Werror` 把警告变成错误，那么你可以确保你不会意外地错过一个警告。

你也可以使用 Pylint 项目和它的 `--py3k` 标志来提示你的代码，当你的代码开始偏离 Python 3 的兼容性时，就会收到警告。这也避免了你不得不定期在你的代码上运行 Modernize 或 Futurize 来捕捉兼容性的退步。这确实要求你只支持 Python 2.7 和 Python 3.4 或更新的版本，因为这是 Pylint 支持的最小 Python 版本。

## 2.7 检查哪些依赖性会阻碍你的过渡

在你使你的代码与 Python 3 兼容 ** 之后 **，你应该开始关心你的依赖关系是否也被移植了。caniusepython3 项目的建立是为了帮助你确定哪些项目——直接或间接地——阻碍了你对 Python 3 的支持。它既有一个命令行工具，也有一个在 https://caniusepython3.com 的网页界面。

该项目还提供了一些代码，你可以将其集成到你的测试套件中，这样，当你不再有依赖关系阻碍你使用 Python 3 时，你将有一个失败的测试。这使你不必手动检查你的依赖性，并在你可以开始在 Python 3 上运行时迅速得到通知。

## 2.8 更新你的 `setup.py` 文件以表示对 Python 3 的兼容

一旦你的代码在 Python 3 下工作，你应该更新你 `setup.py` 中的分类器，使其包含 `Programming Language :: Python :: 3` 并不指定单独的 Python 2 支持。这将告诉使用你的代码的人，你支持 Python 2 **和** 3。理想情况下，你也希望为你现在支持的 Python 的每个主要/次要版本添加分类器。

## 2.9 使用持续集成以保持兼容

一旦你能够完全在 Python 3 下运行，你将会希望确保你的代码在 Python 2 和 3 下总是能够正常工作。在多个 Python 解释器下运行测试的最佳工具可能就是 tox。然后你可以将 tox 集成到你的持续集成系统中，这样你就不会意外地破坏对 Python 2 或 3 的支持。

你可能还想在 Python 3 解释器中使用 `-bb`` 标志，以便在你将 `bytes` 与 `string` 或 `bytes` 与 `int` 进行比较时触发一个异常（后者从 `Python 3.5` 开始可用）。默认情况下，类型不同的比较只是简单地返回``False`，但是如果你在文本/二进制数据处理或字节的索引分离中犯了一个错误，你就不容易发现这个错误。当这些类型的比较发生时，这个标志会触发一个异常，使错误更容易被发现。

## 2.10 考虑使用可选的静态类型检查

另一个帮助移植你的代码的办法是在你的代码上使用 static type checker 如 mypy 或 pytype。这些工具可以用来分析你的代码，就像它在 Python 2 中运行一样，然后你可以第二次运行这个工具，就像你的代码在 Python 3 中运行一样。通过像这样两次运行静态类型检查器，你可以发现你在不同的 Python 版本中是否错误地使用了二进制数据类型。如果你在你的代码中添加了可选的类型提示，你还可以明确说明你的 API 是使用文本还是二进制数据，这有助于确保在两个版本的 Python 中所有的功能都符合预期。.