
将 Python 2 代码迁移到 Python 3

发布 3.10.18

Guido van Rossum
and the Python development team

七月 08, 2025

Python Software Foundation
Email: docs@python.org

Contents

1	简要说明	2
2	详情	2
2.1	删除对 Python 2.6 及更早版本的支持	2
2.2	确保你在你的 <code>setup.py</code> 文件中指定适当的版本支持	3
2.3	良好的测试覆盖率	3
2.4	了解 Python 2 和 3 之间的区别	3
2.5	更新代码	3
2.6	防止兼容性退步	6
2.7	检查哪些依赖性会阻碍你的过渡	6
2.8	更新你的 <code>setup.py</code> 文件以表示对 Python 3 的兼容	6
2.9	使用持续集成以保持兼容	6
2.10	考虑使用可选的静态类型检查	7

作者 Brett Cannon

摘要

Python 3 是 Python 的未来，但 Python 2 仍处于活跃使用阶段，最好让您的项目在两个主要版本的 Python 上都可用。本指南旨在帮助您了解如何最好地同时支持 Python 2 和 3。

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

如果你想了解核心 Python 开发者对于 Python 3 的出现有何看法，你可以阅读 Nick Coghlan 的 [Python 3 Q & A](#) 或 Brett Cannon 的 [为什么要有 Python 3](#)。

关于迁移的帮助，你可以查看存档的 [python-porting](#) 邮件列表。

1 简要说明

为了使你的项目以一份源代码与 Python 2/3 兼容，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 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. 一旦你的依赖性不再阻碍你，使用持续集成来确保你与 Python 2 和 3 保持兼容 (`tox` 可以帮助对多个版本的 Python 进行测试；`python -m pip install tox`)
8. 考虑使用可选的静态类型检查，以确保你的类型用法在 Python 2 和 3 中都适用 (例如，使用 `mypy` 来检查你在 Python 2 和 Python 3 中的类型；`python -m pip install mypy`)。

注解：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

同时支持 Python 2 和 3 的一个关键点是，你可以从 **** 今天 **** 开始！即使你的依赖关系还不支持 Python 3，也不意味着你不能现在就对你的代码进行现代化改造以支持 Python 3。支持 Python 3 所需的大多数变化都会使代码更干净，甚至在 Python 2 的代码中也会使用更新的做法。

另一个关键点是，使你的 Python 2 代码现代化以支持 Python 3 在很大程度上是为你自动化的。虽然由于 Python 3 清晰区分了文本数据与二进制数据，你可能必须做出一些 API 决定。但现在已经为你完成了大部分底层的工作，因此至少可以立即从自动化变化中受益。

当你继续阅读关于迁移你的代码以同时支持 Python 2 和 3 的细节时，请牢记这些关键点。

2.1 删除对 Python 2.6 及更早版本的支持

虽然你可以让 Python 2.5 与 Python 3 一起工作，但如果你只需要与 Python 2.7 一起工作，那就会 **更加容易**。如果放弃 Python 2.5 不是一种选择，那么 `six` 项目可以帮助你同时支持 Python 2.5 和 3 (`python -m pip install six`)。不过请注意，本 HOWTO 中列出的几乎所有项目都有可能不再可用。

如果你能够跳过 Python 2.5 和更早的版本，那么对你的代码所做的必要修改应该看起来和感觉上都是你已经习惯的 Python 代码。在最坏的情况下，你将不得不使用一个函数而不是一个实例方法，或者不得不导入一个函数而不是使用一个内置的函数，但除此之外，整体的转变应该不会让你感到陌生。

但你的目标应该是只支持 Python 2.7。Python 2.6 不再被免费支持，因此也就没有得到错误修复。这意味着 **** 你 **** 必须解决你在 Python 2.6 上遇到的任何问题。本 HOWTO 中提到的一些工具也不支持 Python 2.6 (例如，`Pylint`)，随着时间的推移，这将变得越来越普遍。如果你只支持你必须支持的 Python 版本，那么对你来说会更容易。

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 3 之间的区别

一旦你的代码经过了很好的测试，你就可以开始把你的代码移植到 Python 3 上了！但是为了充分了解你的代码将发生怎样的变化，以及你在编码时要注意什么，你将会想要了解 Python 3 相比 Python 2 的变化。通常来说，两个最好的方法是阅读 Python 3 每个版本的 “What’s New” 文档和 [Porting to Python 3](#) 书（在线免费版本）。还有一个来自 Python-Future 项目的便利 [cheat sheet](#)。

2.5 更新代码

一旦你觉得你知道了 Python 3 与 Python 2 相比有什么不同，就是时候更新你的代码了！你可以选择两种工具来自动移植你的代码：[Futurize](#) 和 [Modernize](#)。你选择哪个工具将取决于你希望你的代码有多像 Python 3。[Futurize](#) 尽力使 Python 3 的习性和做法在 Python 2 中存在，例如，从 Python 3 中回传 `bytes` 类型，这样你就可以在 Python 的主要版本之间实现语义上的平等。另一方面，[Modernize](#) 更加保守，它针对的是 Python 2/3 的子集，直接依靠 [six](#) 来帮助提供兼容性。由于 Python 3 是未来，最好考虑 [Futurize](#)，以开始适应 Python 3 所引入的、你还不习惯的任何新做法。

无论你选择哪种工具，它们都会更新你的代码，使之在 Python 3 下运行，同时与你开始使用的 Python 2 版本保持兼容。根据你想要的保守程度，你可能想先在你的测试套件上运行该工具，并目测差异以确保转换的准确性。在你转换了你的测试套件并验证了所有的测试仍然如期通过后，你就可以转换你的应用程序代码了，因为你知道任何测试失败都是转译的错误。

不幸的是，这些工具不能自动地使你的代码在 Python 3 下工作，因此有一些东西你需要手动更新以获得对 Python 3 的完全支持（这些步骤在不同的工具中是必要的）。阅读你选择使用的工具的文档，看看它默认修复了什么，以及它可以选择做什么，以了解哪些将（不）为你修复，哪些你可能必须自己修复（例如，在 [Modernize](#) 中使用 `io.open()` 覆写内置的 `open()` 函数默认是关闭的）。不过幸运的是，只有几件需要注意的事情，可以说是大问题，如果不注意的话，可能很难调试。

除法

在 Python 3 中, `5 / 2 == 2.5` 而不是 `2`; 所有 `int` 数值之间的除法都会产生一个 `float`、这个变化实际上从 2002 年发布的 Python 2.2 开始就已经计划好了。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`, 或者在运行解释器时使用 `-Q` 标志。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. 根据需要使用任何除法运算符, 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, 一个用户定义的类型用 `/` 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`。

为了使文本和二进制数据之间的区别更加清晰和明显, Python 3 做了大多数在互联网时代创建的语言所做的事情, 使文本和二进制数据成为不能盲目混合在一起的不同类型 (Python 早于互联网的广泛使用)。对于任何只处理文本或只处理二进制数据的代码, 这种分离并不构成问题。但是对于必须处理这两种数据的代码来说, 这确实意味着你现在可能必须关心你何时使用文本或二进制数据, 这就是为什么这不能完全自动化迁移。

首先, 你需要决定哪些 API 接受文本, 哪些接受二进制 (由于保持代码工作的难度, **强烈**建议你不要设计同时接受两种数据的 API; 如前所述, 这很难做得好)。在 Python 2 中, 这意味着要确保处理文本的 API 能够与 `unicode` 一起工作, 而处理二进制数据的 API 能够与 Python 3 中的 `bytes` 类型一起工作 (在 Python 2 中是 `str` 的一个子集, 在 Python 2 中作为 `bytes` 类型的别名)。通常最大的问题是意识到哪些方法同时存在于 Python 2 和 3 的哪些类型上 (对于文本来说, Python 2 中是 `unicode`, Python 3 中是 `str`, 对于二进制来说, Python 2 中是 `str/bytes`, Python 3 中是 `bytes`)。下表列出了每个数据类型在 Python 2 和 3 中的 **独特** 的方法 (例如, `decode()` 方法在 Python 2 或 3 中的等价二进制数据类型上是可用的, 但是它不能在 Python 2 和 3 之间被文本数据类型一致使用, 因为 Python 3 中的 `str` 没有这个方法)。请注意, 从 Python 3.5 开始, `__mod__` 方法被添加到 `bytes` 类型中。

文本数据	二进制数据
	<code>decode</code>
<code>encode</code>	
<code>format</code>	
<code>isdecimal</code>	
<code>isnumeric</code>	

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头是代表文本还是二进制数据。你应该给任何呈现二进制数据的字符符号添加一个 `b` 前缀。对于文本, 你应该给文本字面添加一个 `u` 前缀。(有一个 `__future__` 导入来强制所有未指定的字头为 `Unicode`, 但实际使用情况表明它并不像给所有字头显式添加一个 `b` 或 `u` 前缀那样有效)

作为这种二分法的一部分，你还需要小心打开文件。除非你一直在 Windows 上工作，否则你有可能在打开二进制文件时没有一直费心地添加 b 模式（例如，用 rb 进行二进制读取）。在 Python 3 下，二进制文件和文本文件显然是不同的，而且是相互不兼容的；详见 io 模块。因此，你必须 **** 决定 **** 一个文件是用于二进制访问（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）。你还应该使用 io.open() 来打开文件，而不是内置的 open() 函数，因为 io 模块从 Python 2 到 3 是一致的，而内置的 open() 函数则不是（在 Python 3 中它实际上是 io.open()）。不要理会使用 codecs.open() 的过时做法，因为这只是为了保持与 Python 2.5 的兼容性。

在 Python 2 和 3 中，str 和 bytes 的构造函数对相同的参数有不同的语义。在 Python 2 中，传递一个整数给 bytes，你将得到整数的字符串表示：bytes(3) == '3'。但是在 Python 3 中，一个整数参数传递给 bytes 将给你一个与指定的整数一样长的 bytes 对象，其中充满了空字节：bytes(3) == b'\x00\x00\x00'。当把 bytes 对象传给 str 时，类似的担心是必要的。在 Python 2 中，你只是又得到了该 bytes 对象：str(b'3') == b'3'。但是在 Python 3 中，你得到 bytes 对象的字符串表示：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，你将希望确保你的代码总是在 Python 2 和 3 下工作。在多个 Python 解释器下运行测试的最好工具可能是 `tox`。然后你可以将 `tox` 与你的持续集成系统集成，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。

基本上就是这样了! 在这一点上, 你的代码库同时与 Python 2 和 3 兼容。你的测试也将被设置为不会意外地破坏 Python 2 或 3 的兼容性, 无论你在开发时通常在哪个版本下运行测试。

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方法是在你的代码上使用静态类型检查器, 如 `mypy` 或 `pytype`。这些工具可以用来分析你的代码, 就像它在 Python 2 下运行一样, 然后你可以第二次运行这个工具, 就像你的代码在 Python 3 下运行一样。通过像这样两次运行静态类型检查器, 你可以发现你是否错误地使用了二进制数据类型, 例如在 Python 的一个版本中与另一个版本相比。如果你在你的代码中添加了可选的类型提示, 你也可以明确说明你的 API 是使用文本数据还是二进制数据, 这有助于确保在两个版本的 Python 中所有的功能都符合预期。