

---

# Portando código de Python 2 a Python 3

Versión 3.10.18

Guido van Rossum  
and the Python development team

julio 08, 2025

Python Software Foundation  
Email: docs@python.org

## Índice general

<b>1</b>	<b>La breve explicación</b>	<b>2</b>
<b>2</b>	<b>Detalles</b>	<b>2</b>
2.1	Compatibilidad con Python 2.6 y versiones anteriores	2
2.2	Asegúrese de especificar el soporte de versión adecuado en su archivo <code>setup.py</code>	3
2.3	Tener una buena cobertura de prueba	3
2.4	Aprende las diferencias entre Python 2 & 3	3
2.5	Actualiza tu código	3
2.6	Evitar regresiones de compatibilidad	6
2.7	Compruebe qué dependencias bloquean la transición	7
2.8	Actualice su archivo <code>setup.py</code> para denotar compatibilidad con Python 3	7
2.9	Utilice la integración continua para seguir siendo compatible	7
2.10	Considere la posibilidad de usar la comprobación de tipos estáticos opcionales	7

---

**autor** Brett Cannon

### Resumen

Dado que Python 3 es el futuro de Python mientras Python 2 todavía está en uso activo, es bueno tener su proyecto disponible para ambas versiones principales de Python. Esta guía está diseñada para ayudarle a averiguar la mejor manera de admitir Python 2 y 3 simultáneamente.

Si está buscando portar un módulo de extensión en lugar de código Python puro, consulte [cporting-howto](#).

Si desea leer la opinión de un desarrollador central de Python sobre por qué Python 3 nació, puede leer las [Python 3 Q & A](#) de Nick Coghlan o el artículo de Brett Cannon [Why Python 3 exists](#).

Para obtener ayuda con la portabilidad, puede ver la lista de correo archivada de [python-porting](#).

# 1 La breve explicación

Para que su proyecto sea compatible con Python 2/3 de una sola fuente, los pasos básicos son:

1. Sólo preocúpate por admitir Python 2.7
2. Asegúrese de tener una buena cobertura de prueba ([coberturas.py](#) puede ayudar; `python -m pip install coverage`)
3. Aprende las diferencias entre Python 2 & 3
4. Utilice [Futurize](#) (o [Modernize](#)) para actualizar su código (por ejemplo, `python -m pip install future`)
5. Use [Pylint](#) para asegurarse de que no retrocede en su compatibilidad con Python 3 (`python -m pip install pylint`)
6. Use [caniusepython3](#) para averiguar cuáles de sus dependencias están bloqueando el uso de Python 3 (`python -m pip install caniusepython3`)
7. Una vez que sus dependencias ya no lo bloqueen, use la integración continua para asegurarse de que sigue siendo compatible con Python 2 y 3 ([tox](#) puede ayudar a probar contra múltiples versiones de Python; `python -m pip install tox`)
8. Considere usar la verificación de tipo estática opcional para asegurarse de que su uso de tipo funcione tanto en Python 2 como en 3 (por ejemplo, use [mypy](#) para verificar su escritura en Python 2 y Python 3; `python -m pip install mypy`).

---

**Nota:** Nota: El uso de `python -m pip install` garantiza que el `pip` que invoca es el que está instalado para el Python actualmente en uso, ya sea un `pip` de todo el sistema o uno instalado dentro de un entorno virtual.

---

## 2 Detalles

Un punto clave sobre el soporte de Python 2 & 3 simultáneamente es que se puede empezar **hoy**! Incluso si sus dependencias no son compatibles con Python 3 todavía eso no significa que no puede modernizar el código **ahora** para admitir Python 3. La mayoría de los cambios necesarios para admitir Python 3 conducen a código más limpio utilizando prácticas más recientes incluso en código Python 2.

Otro punto clave es que la modernización del código de Python 2 para que también admita Python 3 está en gran medida automatizada para usted. Si bien es posible que tenga que tomar algunas decisiones de API gracias a python 3 aclarando los datos de texto frente a los datos binarios, el trabajo de nivel inferior ahora se realiza principalmente por usted y, por lo tanto, al menos puede beneficiarse de los cambios automatizados inmediatamente.

Tenga en cuenta esos puntos clave mientras lee sobre los detalles de la migración del código para admitir Python 2 & 3 simultáneamente.

### 2.1 Compatibilidad con Python 2.6 y versiones anteriores

Si bien puede hacer que Python 2.5 funcione con Python 3, es **mucho** más fácil si solo tiene que trabajar con Python 2.7. Si eliminar Python 2.5 no es una opción, entonces el proyecto [six](#) puede ayudarlo a admitir Python 2.5 y 3 simultáneamente (`python -m pip install six`). Sin embargo, tenga en cuenta que casi todos los proyectos enumerados en este **COMO** no estarán disponibles para usted.

Si puede omitir Python 2.5 y versiones anteriores, los cambios necesarios en el código deben seguir pareciendo código Python idiomático. En el peor de los casos tendrá que utilizar una función en lugar de un método en algunos casos o tendrá que importar una función en lugar de usar una integrada, pero de lo contrario la transformación general no debería sentirse ajena a usted.

Pero usted debe apuntar a sólo apoyar Python 2.7. Python 2.6 ya no se admite libremente y, por lo tanto, no recibe correcciones de errores. Esto significa que **usted** tendrá que solucionar cualquier problema que encuentre con Python 2.6. También hay algunas herramientas mencionadas en este HOWTO que no son compatibles con Python 2.6 (por ejemplo, [Pylint](#)), y esto se volverá más común a medida que pasa el tiempo. Simplemente será más fácil para usted si sólo admite las versiones de Python que tiene que admitir.

## 2.2 Asegúrese de especificar el soporte de versión adecuado en su archivo `setup.py`

En su archivo `setup.py` debe tener el [trove classifier](#) adecuado especificando qué versiones de Python admite. Como su proyecto no es compatible con Python 3, al menos debe tener `Programming Language :: Python :: 2 :: Only` especificado. Idealmente también debe especificar cada versión principal/menor de Python que admita, por ejemplo, `Programming Language :: Python :: 2.7`.

## 2.3 Tener una buena cobertura de prueba

Una vez que tenga su código compatible con la versión más antigua de Python 2 que desee, querrá asegurarse de que su conjunto de pruebas tenga una buena cobertura. Una buena regla general es que si desea tener la suficiente confianza en su conjunto de pruebas, cualquier falla que aparezca después de que las herramientas reescriban su código son errores reales en las herramientas y no en su código. Si desea un número al que apuntar, intente obtener una cobertura superior al 80% (y no se sienta mal si le resulta difícil obtener una cobertura superior al 90%). Si aún no tiene una herramienta para medir la cobertura de la prueba, se recomienda [cover.py](#).

## 2.4 Aprende las diferencias entre Python 2 & 3

Una vez que tenga su código bien probado, ¡está listo para comenzar a migrar su código a Python 3! Pero para comprender completamente cómo va a cambiar el código y qué desea tener en cuenta mientras codifica, querrá aprender qué cambios produce Python 3 en términos de Python 2. Típicamente las dos mejores maneras de hacer eso es leer la documentación «What's New» para cada versión de Python 3 y el libro [Porting to Python 3](#) (que es gratis en línea). También hay una práctica [cheat sheet](#) del proyecto Python-Future. </whatsnew-index>.

## 2.5 Actualiza tu código

Una vez que sientas que sabes lo que es diferente en Python 3 en comparación con Python 2, ¡es hora de actualizar tu código! Puede elegir entre dos herramientas para migrar el código automáticamente: [Futurize](#) y [Modernize](#). La herramienta que elija dependerá de la cantidad similar a Python 3 que desea que sea el código. [Futurize](#) hace todo lo posible para que Python 3 modismos y prácticas existan en Python 2, por ejemplo, backporting el tipo `bytes` de Python 3 para que tenga paridad semántica entre las versiones principales de Python. [Modernize](#), por otro lado, es más conservador y se dirige a un subconjunto de Python 2/3 de Python, basándose directamente en [six](#) para ayudar a proporcionar compatibilidad. Como Python 3 es el futuro, podría ser mejor considerar [Futurize](#) para comenzar a adaptarse a cualquier nueva práctica que Python 3 introduce a la que aún no está acostumbrado.

Independientemente de la herramienta que elija, actualizarán el código para que se ejecute en Python 3 mientras se mantienen compatibles con la versión de Python 2 con la que comenzó. Dependiendo de lo conservador que desee ser, es posible que desee ejecutar la herramienta sobre el conjunto de pruebas primero e inspeccionar visualmente la diferencia para asegurarse de que la transformación es precisa. Después de transformar el conjunto de pruebas y comprobar que todas las pruebas siguen pasando según lo esperado, puede transformar el código de la aplicación sabiendo que cualquier prueba que falle es un error de traducción.

Desafortunadamente, las herramientas no pueden automatizar todo para que su código funcione bajo Python 3 y por lo que hay un puñado de cosas que tendrá que actualizar manualmente para obtener soporte completo de Python 3 (cuáles de estos pasos son necesarios varían entre las herramientas). Lea la documentación de la herramienta que elige utilizar para ver lo que corrige de forma predeterminada y lo que puede hacer opcionalmente para saber lo que (no) se fijará para usted y lo que puede tener que corregir por su cuenta (por ejemplo, usando `io.open()` sobre la función incorporada `open()` está desactivada por defecto en [Modernizar](#)). Afortunadamente, sin embargo, sólo hay

un par de cosas a tener en cuenta por las cuales se pueden considerar grandes problemas que pueden ser difíciles de depurar si no se observan.

## División

En Python 3, `5 / 2 == 2.5` y no `2`; toda división entre los valores `int` da lugar a un `float`. Este cambio ha sido planeado desde Python 2.2, que fue lanzado en 2002. Desde entonces, se ha alentado a los usuarios a añadir `from __future__ import division` a todos y cada uno de los archivos que utilizan los operadores `/` y `//` o que ejecuten el intérprete con el indicador `-Q`. Si no ha estado haciendo esto, entonces tendrá que ir a través de su código y hacer dos cosas:

1. Añadir `from __future__ import division` a sus archivos
2. Actualice cualquier operador de división según sea necesario para utilizar `//` para usar la división de suelo o continuar usando `/` y esperar un número flotante

La razón por la que `/` no se traduce simplemente a `//` automáticamente es que si un objeto define un método `__truediv__` pero no `__floordiv__` entonces su código comenzaría a fallar (por ejemplo, una clase definida por el usuario que utiliza `/` para significar alguna operación pero no `//` para la misma cosa o en absoluto).

## Texto frente a datos binarios

En Python 2 puede usar el tipo `str` tanto para texto como para datos binarios. Desafortunadamente, esta confluencia de dos conceptos diferentes podría conducir a código frágil que a veces funcionaba para cualquier tipo de datos, a veces no. También podría dar lugar a API confusas si las personas no declaraban explícitamente que algo que aceptaba `str` aceptaba datos binarios o de texto en lugar de un tipo específico. Esto complicó la situación especialmente para cualquier persona que admita varios idiomas, ya que las API no se molestarían explícitamente en admitir explícitamente `Unicode` cuando reclamaban compatibilidad con datos de texto.

Para hacer la distinción entre texto y datos binarios más claros y pronunciados, Python 3 hizo lo que la mayoría de los lenguajes creados en la era de Internet han hecho y ha hecho texto y datos binarios distintos tipos que no se pueden mezclar ciegamente (Python es anterior al acceso generalizado a Internet). Para cualquier código que se ocupe solo de texto o solo de datos binarios, esta separación no plantea un problema. Pero para el código que tiene que lidiar con ambos, significa que es posible que tenga que preocuparse ahora cuando está utilizando texto en comparación con los datos binarios, por lo que esto no se puede automatizar por completo.

Para empezar, tendrá que decidir qué API toman texto y cuáles toman binario (es **altamente** recomendado no diseñar API que pueden tomar ambos debido a la dificultad de mantener el código funcionando; como se indicó anteriormente es difícil hacerlo bien). En Python 2 esto significa asegurarse de que las API que toman texto pueden trabajar con `unicode` y las que funcionan con datos binarios funcionan con el tipo `bytes` de Python 3 (que es un subconjunto de `str` en Python 2 y actúa como un alias para `bytes` tipo en Python 2). Por lo general, el mayor problema es darse cuenta de qué métodos existen en qué tipos en Python 2 y 3 simultáneamente (para el texto que es `Unicode` en Python 2 y `str` en Python 3, para binario que es `str/bytes` en Python 2 y `bytes` en Python 3). En la tabla siguiente se enumeran los métodos **unicos** de cada tipo de datos en Python 2 y 3 (por ejemplo, el método `decode()` se puede utilizar en el tipo de datos binarios equivalente en Python 2 o 3, pero no puede ser utilizado por el tipo de datos textuales consistentemente entre Python 2 y 3 porque `str` en Python 3 no tiene el método). Tenga en cuenta que a partir de Python 3.5 se agregó el método `__mod__` al tipo `bytes`.

Datos de texto	Datos binarios
	<code>decode</code>
<code>encode</code>	
<code>format</code>	
<code>isdecimal</code>	
<code>isnumeric</code>	

La creación de la distinción más fácil de controlar se puede realizar mediante la codificación y decodificación entre datos binarios y texto en el borde del código. Esto significa que cuando reciba texto en datos binarios, debe decodificarlo inmediatamente. Y si el código necesita enviar texto como datos binarios, codificarlo lo más tarde

posible. Esto permite que el código funcione solo con texto internamente y, por lo tanto, elimina tener que realizar un seguimiento del tipo de datos con los que está trabajando.

El siguiente problema es asegurarse de saber si los literales de cadena en el código representan texto o datos binarios. Debe agregar un prefijo `b` a cualquier literal que presente datos binarios. Para el texto debe agregar un prefijo `u` al literal de texto. (hay una importación `__future__` para forzar que todos los literales no especificados sean Unicode, pero el uso ha demostrado que no es tan eficaz como agregar un prefijo `b` o `u` a todos los literales explícitamente)

Como parte de esta dicotomía también hay que tener cuidado con la apertura de archivos. A menos que haya estado trabajando en Windows, existe la posibilidad de que no siempre se haya molestado en agregar el modo `b` al abrir un archivo binario (por ejemplo, `rb` para la lectura binaria). En Python 3, los archivos binarios y los archivos de texto son claramente distintos y mutuamente incompatibles; ver el módulo `io` para más detalles. Por lo tanto, **debe** tomar una decisión de si un archivo se utilizará para el acceso binario (permitiendo que los datos binarios se lean y/o escriban) o el acceso textual (permitiendo que los datos de texto sean leídos y/o escritos). También debe utilizar `io.open()` para abrir archivos en lugar de la función incorporada `open()` como el módulo `io` es consistente de Python 2 a 3, mientras que la función incorporada `open()` no es (en Python 3 es en realidad `io.open()`). No se moleste con la práctica obsoleta de usar `codecs.open()` ya que sólo es necesario para mantener la compatibilidad con Python 2.5.

Los constructores de `str` y `bytes` tienen una semántica diferente para los mismos argumentos entre Python 2 y 3. Pasar un entero a `bytes` en Python 2 le dará la representación de cadena de texto del entero: `bytes(3) == '3'`. Pero en Python 3, un argumento entero para `bytes` le dará un objeto `bytes` siempre y cuando el entero especificado, lleno de bytes nulos: `bytes(3) == b'\x00\x00\x00'`. Una preocupación similar es necesaria cuando se pasa un objeto `bytes` a `str`. En Python 2, solo se obtiene el objeto `bytes`: `str(b'3') == b'3'`. Pero en Python 3 se obtiene la representación de cadena de texto del objeto `bytes`: `str(b'3') == "b'3'"`.

Por último, la indexación de datos binarios requiere un control cuidadoso (el corte **no** requiere ningún control especial). En Python 2, `b'123'[1] == b'2'` mientras que en Python 3 `b'123'[1] == 50`. Dado que los datos binarios son simplemente una colección de números binarios, Python 3 retorna el valor entero para el byte en el que indexa. Pero en Python 2, ya que `bytes == str`, la indexación retorna un segmento de `bytes` de un solo elemento. El proyecto `six` tiene una función denominada `six.indexbytes()` que devolverá un entero como en Python 3: `six.indexbytes(b'123', 1)`.

Para resumir:

1. Decida cuál de sus API toma texto y cuáles toman datos binarios
2. Asegúrese de que el código que funciona con texto también funciona con `unicode` y el código para datos binarios funciona con `bytes` en Python 2 (consulte la tabla anterior para los métodos que no puede usar para cada tipo)
3. Marque todos los literales binarios con un prefijo `b`, literales textuales con un prefijo `u`
4. Descodificar datos binarios en texto tan pronto como sea posible, codificar texto como datos binarios tan tarde como sea posible
5. Abra los archivos con `io.open()` y asegúrese de especificar el modo `b` cuando sea apropiado
6. Tenga cuidado al indexar en datos binarios

## Utilice la detección de funciones en lugar de la detección de versiones

Inevitablemente tendrá código que tiene que elegir qué hacer en función de qué versión de Python se está ejecutando. La mejor manera de hacerlo es con la detección de características de si la versión de Python en la que se ejecuta es compatible con lo que necesita. Si por alguna razón eso no funciona, entonces usted debe hacer que la comprobación de la versión sea contra Python 2 y no Python 3. Para ayudar a explicar esto, veamos un ejemplo.

Supongamos que necesita acceso a una característica de `importlib` que está disponible en la biblioteca estándar de Python desde Python 3.3 y disponible para Python 2 a través de `importlib2` en PyPI. Es posible que tenga la tentación de escribir código para acceder, por ejemplo, al módulo `importlib.abc` haciendo lo siguiente:

```
import sys

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

El problema con este código es ¿qué sucede cuando sale Python 4? Sería mejor tratar Python 2 como el caso excepcional en lugar de Python 3 y asumir que las futuras versiones de Python serán más compatibles con Python 3 que Python 2:

```
import sys

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

La mejor solución, sin embargo, es no hacer ninguna detección de versiones en absoluto y en su lugar confiar en la detección de características. Esto evita cualquier problema potencial de conseguir la detección de la versión incorrecta y le ayuda a mantenerse compatible con el futuro:

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

## 2.6 Evitar regresiones de compatibilidad

Una vez que haya traducido completamente el código para que sea compatible con Python 3, querrá asegurarse de que el código no retroceda y deje de funcionar bajo Python 3. Esto es especialmente cierto si tiene una dependencia que le está bloqueando para que no se ejecute realmente en Python 3 en este momento.

Para ayudar a mantenerse compatible, los módulos nuevos que cree deben tener al menos el siguiente bloque de código en la parte superior del mismo:

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

También puede ejecutar Python 2 con el indicador `-3` para recibir una advertencia sobre varios problemas de compatibilidad que el código desencadena durante la ejecución. Si convierte las advertencias en errores con `-Werror`, puede asegurarse de que no se pierda accidentalmente una advertencia.

También puede usar el proyecto de [Pylint](#) y su indicador `--py3k` para lintar el código para recibir advertencias cuando el código comienza a desviarse de la compatibilidad con Python 3. Esto también evita que tenga que ejecutar [Modernize](#) o [Futurize](#) sobre el código con regularidad para detectar las regresiones de compatibilidad. Esto requiere que solo admita Python 2.7 y Python 3.4 o posterior, ya que es la compatibilidad mínima de la versión mínima de Python de Pylint.

## 2.7 Compruebe qué dependencias bloquean la transición

**Después** de que haya hecho que su código sea compatible con Python 3, debe empezar a preocuparse por si sus dependencias también se han portado. El proyecto [caniusepython3](https://caniusepython3.com) se creó para ayudarle a determinar qué proyectos – directa o indirectamente – le impiden admitir Python 3. Hay una herramienta de línea de comandos, así como una interfaz web en <https://caniusepython3.com>.

El proyecto también proporciona código que puede integrar en el conjunto de pruebas para que tenga una prueba con errores cuando ya no tenga dependencias que le impidan usar Python 3. Esto le permite evitar tener que comprobar manualmente sus dependencias y recibir notificaciones rápidamente cuando puede empezar a ejecutarse en Python 3.

## 2.8 Actualice su archivo `setup.py` para denotar compatibilidad con Python 3

Una vez que el código funciona en Python 3, debe actualizar los clasificadores en su `setup.py` para que contenga `Programming Language :: Python :: 3` y no especificar solo compatibilidad con Python 2. Esto le dirá a cualquier persona que use su código que admite Python 2 y 3. Lo ideal es que también desee agregar clasificadores para cada versión principal/menor de Python que ahora admita.

## 2.9 Utilice la integración continua para seguir siendo compatible

Una vez que pueda ejecutar completamente bajo Python 3, querrá asegurarse de que el código siempre funciona en Python 2 y 3. Probablemente la mejor herramienta para ejecutar las pruebas en varios intérpretes de Python es `tox`. A continuación, puede integrar `tox` con su sistema de integración continua para que nunca interrumpa accidentalmente la compatibilidad con Python 2 o 3.

También es posible que desee utilizar el indicador `-bb` con el intérprete de Python 3 para desencadenar una excepción cuando se comparan bytes con cadenas o bytes con un `int` (este último está disponible a partir de Python 3.5). De forma predeterminada, las comparaciones de tipos diferentes simplemente retornan `False`, pero si cometió un error en la separación del control de datos de texto/binario o la indexación en bytes, no encontraría fácilmente el error. Esta marca lanzará una excepción cuando se produzcan este tipo de comparaciones, lo que hace que el error sea mucho más fácil de rastrear.

¡Y eso es sobre todo! En este punto, la base de código es compatible con Python 2 y 3 simultáneamente. Las pruebas también se configurarán para que no interrumpa accidentalmente la compatibilidad de Python 2 o 3, independientemente de la versión en la que ejecute normalmente las pruebas durante el desarrollo.

## 2.10 Considere la posibilidad de usar la comprobación de tipos estáticos opcionales

Otra forma de ayudar a transferir el código es usar un comprobador de tipos estáticos como `mypy` o `pytype` en el código. Estas herramientas se pueden utilizar para analizar el código como si se estuviera ejecutando en Python 2, puede ejecutar la herramienta por segunda vez como si el código se ejecutara en Python 3. Al ejecutar un comprobador de tipos estáticos dos veces como este, puede descubrir si, por ejemplo, está usando incorrectamente el tipo de datos binarios en una versión de Python en comparación con otra. Si agrega sugerencias de tipo opcionales al código, también puede indicar explícitamente si las API usan datos textuales o binarios, lo que ayuda a asegurarse de que todo funciona según lo esperado en ambas versiones de Python.