
파이썬 2 코드를 파이썬 3으로 이식하기

출시 버전 3.10.16

Guido van Rossum
and the Python development team

12월 07, 2024

Contents

1	짧은 설명	2
2	세부 사항	2
2.1	파이썬 2.6 이하에 대한 지원을 중단하십시오	2
2.2	setup.py 파일에서 올바른 버전 지원을 지정했는지 확인하십시오	3
2.3	양호한 테스트 커버리지를 갖추십시오	3
2.4	파이썬 2와 3의 차이점을 공부하십시오	3
2.5	여러분의 코드를 갱신하십시오	3
2.6	호환성 회귀를 방지하십시오	6
2.7	전환을 막는 의존성을 확인하십시오	6
2.8	파이썬 3 호환성을 나타내도록 setup.py 파일을 갱신하십시오	6
2.9	지속적인 통합을 통해 호환성을 유지하십시오	7
2.10	선택적 정적 형 검사 사용을 고려하십시오	7

저자 Brett Cannon

요약

파이썬 3이 파이썬의 미래이지만 파이썬 2가 여전히 활발하게 사용되고 있기 때문에, 두 가지 주요 파이썬 릴리스에서 프로젝트를 사용할 수 있게 하는 것이 좋습니다. 이 안내서는 파이썬 2와 3을 동시에 지원하는 가장 좋은 방법을 찾는 데 도움을 주려고 합니다.

순수 파이썬 코드 대신 확장 모듈을 이식하려고 한다면 [cporting-howto](#)를 참조하십시오.

파이썬 3이 왜 등장했는지에 대한 핵심 파이썬 개발자의 글을 읽고 싶다면, Nick Coghlan의 [Python 3 Q & A](#)나 Brett Cannon의 [Why Python 3 exists](#)를 추천합니다.

For help with porting, you can view the archived [python-porting](#) mailing list.

1 짧은 설명

프로젝트를 단일 소스 파이썬 2/3 호환으로 만들려면, 기본 단계는 다음과 같습니다:

1. 파이썬 2.7 지원만 신경 쓰십시오
2. 테스트 커버리지가 양호한지 확인하십시오 (`coverage.py`가 도움이 될 수 있습니다; `python -m pip install coverage`)
3. 파이썬 2와 3의 차이점을 공부하십시오
4. `Futurize`(또는 `Modernize`)를 사용하여 코드를 갱신하십시오 (예를 들어 `python -m pip install future`)
5. `Pylint`를 사용하여 파이썬 3 지원의 회귀 테스트가 실패하지 않도록 점검하십시오 (`python -m pip install pylint`)
6. `caniusepython3`를 사용하여 파이썬 3 사용을 막는 의존성 (dependencies)을 찾으십시오 (`python -m pip install caniusepython3`)
7. 일단 의존성이 더는 여러분을 막지 않으면, 지속적인 통합을 사용하여 파이썬 2 & 3과의 호환성을 유지하십시오 (`tox`는 여러 버전의 파이썬에서 테스트하는 데 도움이 됩니다; `python -m pip install tox`)
8. 선택적으로 정적 형 검사를 사용하여 형 사용이 파이썬 2와 3에서 모두 작동하는지 확인하십시오 (예를 들어 `mypy`를 사용하여 파이썬 2와 파이썬 3 모두에서 형 사용을 검사하십시오; `python -m pip install mypy`).

참고: 참고: `python -m pip install`를 사용하면 호출하는 `pip`가 시스템 전체 `pip`이든 가상 환경 내에 설치된 `pip`이든 현재 사용 중인 파이썬을 위해 설치된 `pip`임이 보장됩니다.

2 세부 사항

파이썬 2와 3을 동시에 지원하는 것에 대한 요점은 오늘 시작할 수 있다는 것입니다! 의존성이 아직 파이썬 3을 지원하지 않을 때조차 여러분의 코드를 파이썬 3을 지원하도록 지금 현대화할 수 없다는 뜻은 아닙니다. 파이썬 3을 지원하는데 필요한 대부분의 변경은 파이썬 2 코드에서도 새로운 방법을 사용하여 더 깔끔한 코드를 만듭니다.

또 다른 요점은 파이썬 3도 지원하도록 파이썬 2 코드를 현대화하는 것이 대부분 자동화되어 있다는 것입니다. 여러분이 일부 API 결정을 내려야 할 수도 있지만, 텍스트 데이터와 바이너리 데이터를 명확히 구분하는 파이썬 3 덕분에, 이제 저수준 작업이 대부분 수행되므로 최소한 자동 변경의 이점을 즉시 누릴 수 있습니다.

파이썬 2와 3을 동시에 지원하기 위해 코드를 이식하는 것에 대한 자세한 내용을 읽는 동안 이러한 요점을 명심하십시오.

2.1 파이썬 2.6 이하에 대한 지원을 중단하십시오

파이썬 2.5를 파이썬 3에서 동작하게 만들 수 있지만, 파이썬 2.7만 지원한다면 훨씬 쉽습니다. 파이썬 2.5를 포기하는 것이 옵션이 아니면 `six` 프로젝트를 사용해서 파이썬 2.5와 3을 동시에 지원할 수 있습니다 (`python -m pip install six`). 그러나 이 HOWTO에 나열된 거의 모든 프로젝트를 이용할 수 없다는 것을 알고 계십시오.

파이썬 2.5와 그 이전 버전을 무시할 수 있다면, 코드에 필요한 변경 사항은 계속 관용적인 파이썬 코드처럼 보이고 느껴져야 합니다. 최악의 경우 일부 인스턴스에서 메서드 대신 함수를 사용해야 하거나 내장 함수를 사용하는 대신 함수를 임포트 해야 하지만, 그 외에는 전체적인 변환이 이질적으로 느껴지지 않아야 합니다.

그러나 파이썬 2.7만 지원해야 합니다. 파이썬 2.6은 더는 무료로 지원되지 않아서 버그 수정이 없습니다. 이것은 여러분이 만나는 파이썬 2.6의 문제를 여러분이 해결해야 한다는 뜻입니다. 이 HOWTO에서 언급하는 몇 가지 도구는 파이썬 2.6을 지원하지 않기도 하고 (예를 들어 [Pylint](#)), 시간이 지남에 따라 더 늘어날 것입니다. 지원해야만 하는 파이썬 버전만 지원하는 것이 더 쉬울 것입니다.

2.2 setup.py 파일에서 올바른 버전 지원을 지정했는지 확인하십시오

setup.py 파일에는 여러분이 지원하는 파이썬 버전을 지정하는 적절한 [trove](#) 분류가 있어야 합니다. 여러분의 프로젝트가 아직 파이썬 3을 지원하지 않기 때문에 최소한 `Programming Language :: Python :: 2 :: Only`를 지정해야 합니다. 이상적으로는 지원하는 각 주/부 버전의 파이썬을 지정해야 합니다, 예를 들어 `Programming Language :: Python :: 2.7`.

2.3 양호한 테스트 커버리지를 갖추십시오

일단 여러분이 원하는 가장 오래된 파이썬 2 버전을 지원하는 코드를 확보하면, 테스트 스위트가 양호한 커버리지를 갖는지 확인해야 합니다. 경험 규칙은 도구가 코드를 다시 작성한 후 나타나는 실패가 여러분의 코드가 아니라 도구에 있는 실제 버그라는 확신을 가질 만큼 테스트 스위트를 신뢰할 수 있는 수준입니다. 목표로 할 숫자가 필요하다면, 80% 이상의 커버리지를 시도하십시오 (그리고 90% 이상의 커버리지를 얻기 어려워도 실망하지 마십시오). 테스트 커버리지를 측정하는 도구가 없으면 [coverage.py](#)를 추천합니다.

2.4 파이썬 2와 3의 차이점을 공부하십시오

코드를 잘 테스트했으면, 코드를 파이썬 3으로 이식할 준비가 되었습니다! 그러나 코드가 어떻게 변경되고 코드를 작성하는 동안 무엇을 살펴야 하는지 완전히 이해하려면, 파이썬 3이 파이썬 2에 어떤 변경을 가했는지 배우고 싶을 것입니다. 일반적으로 가장 좋은 두 가지 방법은 각 파이썬 3 릴리스의 “새로운 기능” 문서와 [Porting to Python 3](#) 책(온라인에서 무료로 제공됩니다)을 읽는 것입니다. Python-Future 프로젝트의 편리한 [cheat sheet](#)도 있습니다.

2.5 여러분의 코드를 갱신하십시오

일단 파이썬 3과 파이썬 2의 차이점이 무엇인지 안다고 느낀다면, 코드를 갱신할 차례입니다! 여러분의 코드를 자동으로 이식하는 두 가지 도구 중에서 선택할 수 있습니다: [Futurize](#)와 [Modernize](#). 어떤 도구를 선택하느냐는 여러분의 코드를 얼마나 파이썬 3 답게 만들고 싶은지에 달려 있습니다. [Futurize](#)는 파이썬 3 관용구와 관행을 파이썬 2에 존재하도록 만들기 위해 최선을 다합니다, 예를 들어 파이썬 3의 `bytes` 형을 역 이식하여 파이썬의 주 버전 간에 의미론적 일치가 이루어지도록 합니다. 반면 [Modernize](#)는 더 보수적이며 호환성을 제공하기 위해 [six](#)에 직접 의존하면서 파이썬의 파이썬 2/3 부분 집합을 타겟으로 합니다. 파이썬 3이 미래이기 때문에, 아직 익숙하지 않은 파이썬 3이 도입한 새로운 관행에 적응하기 시작하려면 [Futurize](#)를 고려하는 것이 가장 좋습니다.

어떤 도구를 선택하든, 파이썬 3에서 실행되도록 코드를 갱신하면서 여러분이 시작한 파이썬 2 버전과 호환되도록 유지합니다. 여러분이 얼마나 보수적으로 되고 싶은지에 따라, 먼저 테스트 스위트에 도구를 실행하고 `diff`를 시각적으로 검사하여 변환이 정확한지 확인하고 싶을 수 있습니다. 테스트 스위트를 변환하고 모든 테스트가 여전히 예상대로 통과되는지 확인한 후에는, 실패한 모든 테스트가 변환 실패임을 아는 상태에서 응용 프로그램 코드를 변환할 수 있습니다.

불행히도 도구가 파이썬 3에서 코드가 작동하도록 모든 것을 자동화할 수는 없기 때문에, 완전한 파이썬 3 지원을 얻기 위해 수동으로 갱신해야 하는 몇 가지 사항이 있습니다 (이 단계의 어떤 것이 필요한지는 도구마다 다릅니다). 어떤 것이 자동으로 수정되고 (또는 되지 않고) 어떤 것을 여러분이 직접 수정해야 하는지 알기 위해, 기본적으로 수정되는 것과 선택적으로 수정되는 것에 대해 여러분이 선택한 도구의 설명서를 읽으십시오 (예를 들어 [Modernize](#)에서는 내장 `open()` 함수 대신 `io.open()` 을 사용하는 것은 기본적으로 꺼져 있습니다). 다행히, 주의하지 않으면 디버깅하기 어려운 큰 문제로 간주할 수 있는 주의해야 할 사항은 몇 가지뿐입니다.

나누기

파이썬 3에서, $5 / 2 == 2.5$ 이고 2가 아닙니다; int 값 간의 모든 나눗기는 float가 됩니다. 이 변경은 실제로는 2002년에 릴리스된 파이썬 2.2부터 계획되었습니다. 그때부터 /와 // 연산자를 사용하는 모든 파일에 `from __future__ import division`을 추가하거나 -Q 플래그로 인터프리터를 실행하도록 권장되었습니다. 이 작업을 수행하지 않았으면 코드를 살펴보고 두 가지 작업을 수행해야 합니다:

1. 여러분의 파일에 `from __future__ import division`을 추가하십시오
2. //를 사용하여 정수 나눗셈을 사용하거나 /를 계속 사용하고 float를 기대하도록 필요에 따라 나눗셈 연산자를 갱신하십시오

/가 단순히 //로 자동 변환되지 않는 이유는 객체가 `__truediv__` 메서드를 정의하지만 `__floordiv__`를 정의하지 않으면 코드가 실패하기 시작하기 때문입니다 (예를 들어 /를 사용하여 일부 작업을 나타내지만 //로는 같은 것을 하지 않거나 아예 지원하지 않는 사용자 정의 클래스).

텍스트 대 바이너리 데이터

파이썬 2에서는 텍스트와 바이너리 데이터 모두에 `str` 형을 사용할 수 있습니다. 불행히도 이 두 가지 다른 개념의 합류로 인해 때로는 두 유형의 데이터 모두에서 동작하고 때로는 동작하지 않는 믿을 수 없는 코드가 만들어질 수 있습니다. 하나의 구체적인 형 대신에 `str`로 받아들인 것이 텍스트나 바이너리 데이터 중 어느 것을 받아들이는지 명시적으로 언급하지 않으면 혼란스러운 API가 될 수 있습니다. API가 텍스트 데이터 지원을 주장할 때 명시적으로 unicode를 지원하지 않을 수 있어서 여러 언어를 지원하는 사람들에게는 특히 상황을 복잡하게 만듭니다.

텍스트와 바이너리 데이터의 구별을 보다 명확하고 뚜렷하게 하기 위해, 파이썬 3은 인터넷 시대에 만들어진 대부분의 언어가 수행한 작업을 수행했으며 텍스트와 바이너리 데이터를 맹목적으로 혼합할 수 없는 고유한 형으로 만들었습니다 (파이썬은 인터넷이 널리 퍼지기 전부터 존재해 왔습니다). 텍스트나 바이너리 데이터 어느 한 가지만 처리하는 코드의 경우, 이 분리는 문제를 일으키지 않습니다. 그러나 두 가지를 모두 다뤄야 하는 코드의 경우, 언제 텍스트 데이터를, 언제 바이너리 데이터를 사용해야 할지 이제 신경 써야 할 수 있음을 뜻하고, 이것이 완전히 자동화할 수 없는 이유입니다.

시작하려면, 어떤 API가 텍스트를 취하고 어떤 것이 바이너리를 취할지 결정해야 합니다 (코드가 동작하도록 만드는 어려움 때문에 둘 다 취하는 API를 설계하지 말 것을 강하게 권고합니다; 앞서 언급했듯이 잘하기가 어렵습니다). 파이썬 2에서 이것은 텍스트를 취하는 API가 unicode에서 작동하고 바이너리 데이터로 작동하는 API가 파이썬 3에서 온 bytes 형 (파이썬 2에서 `str`의 부분 집합이며 파이썬 2에서 bytes 형식의 별칭으로 작동합니다)으로 작동하는 것을 의미합니다. 일반적으로 가장 큰 문제는 파이썬 2와 3의 어떤 형에 어떤 메서드가 동시에 존재하는지 인식하는 것입니다 (텍스트의 경우 이것은 파이썬 2에서는 unicode고 파이썬 3에서는 `str`입니다, 바이너리의 경우 이것은 파이썬 2에서는 `str/bytes`이고 파이썬 3에서는 bytes입니다). 다음 표는 파이썬 2와 3을 가로질러 각 데이터형의 고유한 메서드를 나열합니다 (예를 들어 `decode()` 메서드는 파이썬 2와 3의 동등한 바이너리 데이터형에서 사용할 수 있지만, 파이썬 3의 `str`에는 메서드가 없기 때문에 텍스트 데이터형에서는 파이썬 2와 3간에 일관되게 사용할 수 없습니다). 파이썬 3.5부터 `__mod__` 메서드가 bytes 형에 추가되었음에 유의하십시오.

텍스트 데이터	바이너리 데이터
	<code>decode</code>
<code>encode</code>	
<code>format</code>	
<code>isdecimal</code>	
<code>isnumeric</code>	

코드 가장자리에서 바이너리 데이터와 텍스트 간의 인코딩과 디코딩을 함으로써 구별을 더 쉽게 처리할 수 있습니다. 이는 바이너리 데이터로 텍스트를 수신하면 즉시 디코딩해야 함을 의미합니다. 그리고 코드가 텍스트를 바이너리 데이터로 보내야 하면 가능한 한 늦게 인코딩하십시오. 이렇게 하면 코드는 내부적으로 텍스트만 처리하고 작업 중인 데이터의 형을 추적할 필요가 없습니다.

다음 문제는 코드의 문자열 리터럴이 텍스트나 바이너리 데이터 중 어느 것을 나타내는지를 확인하는 것입니다. 바이너리 데이터를 나타내는 모든 리터럴에 `b` 접두사를 추가해야 합니다. 텍스트의 경우 텍스트 리터럴에 `u` 접두사를 추가해야 합니다. (지정되지 않은 모든 리터럴을 유니코드로 강제 적용하는

`__future__` 임포트가 있지만, 그간의 경험으로 보면 모든 리터럴에 `b`나 `u` 접두사를 명시적으로 추가하는 것만큼 효과적이지 않습니다)

이 이분법의 일부로 파일을 열 때도 주의해야 합니다. 윈도우에서 작업해보지 않았다면, 바이너리 파일을 열 때 항상 `b` 모드를 추가하지 (예를 들어 바이너리 읽기를 위한 `rb`) 않았을 수 있습니다. 파이썬 3에서는, 바이너리 파일과 텍스트 파일이 명확하게 구분되고 서로 호환되지 않습니다; 자세한 내용은 `io` 모듈을 참조하십시오. 따라서, 파일을 바이너리 액세스(바이너리 데이터를 읽거나 쓸 수 있도록 합니다)나 텍스트 액세스(텍스트 데이터를 읽거나 쓸 수 있도록 합니다) 중 어느 것으로 사용할지를 반드시 결정해야 합니다. `io` 모듈은 파이썬 2와 3에서 일관성 있지만, 내장 `open()` 함수(파이썬 3에서 실제로는 `io.open()` 입니다)는 그렇지 않기 때문에 내장 `open()` 함수 대신 `io.open()` 을 사용하여 파일을 열어야 합니다. 파이썬 2.5와의 호환성을 유지하는 데만 필요하므로 `codecs.open()` 을 사용하는 오래된 방법은 신경 쓰지 마십시오.

`str`과 `bytes`의 생성자는 모두 파이썬 2와 3 사이에서 같은 인자에 대해 다른 의미가 있습니다. 파이썬 2에서 정수를 `bytes`에 전달하면 정수의 문자열 표현을 줍니다: `bytes(3) == '3'`. 그러나 파이썬 3에서, `bytes`에 대한 정수 인자는 지정된 정수 길이의 널 바이트로 채워진 `bytes` 객체를 줍니다: `bytes(3) == b'\x00\x00\x00'`. `bytes` 객체를 `str`로 전달할 때도 비슷한 주의가 필요합니다. 파이썬 2에서는 단지 `bytes` 객체를 다시 받습니다: `str(b'3') == b'3'`. 그러나 파이썬 3에서는 `bytes` 객체의 문자열 표현을 얻게 됩니다: `str(b'3') == "b'3'"`.

마지막으로, 바이너리 데이터의 인덱싱에는 신중한 처리가 필요합니다 (슬라이싱에는 특별한 처리가 필요하지 않습니다). 파이썬 2에서는, `b'123'[1] == b'2'` 인 반면 파이썬 3에서는 `b'123'[1] == 50` 입니다. 바이너리 데이터는 단순히 바이너리 숫자의 컬렉션이므로, 파이썬 3은 인덱싱한 바이트의 정숫값을 반환합니다. 그러나 파이썬 2에서는 `bytes == str` 때문에, 인덱싱은 한 항목의 `bytes` 슬라이스를 반환합니다. `six` 프로젝트에는 파이썬 2에서처럼 정수를 반환하는 `six.indexbytes()` 라는 함수가 있습니다: `six.indexbytes(b'123', 1)`.

요약하면:

1. 어떤 API가 텍스트를 취하고 어떤 것이 바이너리 데이터를 취하는지 결정하십시오
2. 텍스트로 작동하는 코드가 `unicode`에서도 작동하고 바이너리 데이터를 위한 코드는 파이썬 2에서 `bytes`와 작동하도록 하십시오 (각 형에서 사용할 수 없는 메서드는 위의 표를 참조하십시오)
3. 모든 바이너리 리터럴을 `b` 접두사로 표시하고, 텍스트 리터럴을 `u` 접두사로 표시하십시오
4. 바이너리 데이터를 가능한 한 빨리 텍스트로 디코딩하고, 텍스트를 가능한 한 늦게 바이너리 데이터로 인코딩하십시오
5. `io.open()` 을 사용하여 파일을 열고 적절할 때 `b` 모드를 지정하십시오
6. 바이너리 데이터로 인덱싱할 때 주의하십시오

버전 감지 대신 기능 감지를 사용하십시오

필연적으로 실행 중인 파이썬 버전에 따라 수행할 작업을 선택해야 하는 코드를 갖게 됩니다. 가장 좋은 방법은 실행 중인 파이썬 버전이 필요한 것을 지원하는지에 대한 기능 감지를 사용하는 것입니다. 어떤 이유로 이 방법이 작동하지 않으면 버전 확인을 파이썬 3이 아닌 파이썬 2에 대해 수행해야 합니다. 이를 설명하기 위해, 예제를 살펴보겠습니다.

파이썬 3.3 이후로 파이썬의 표준 라이브러리에서 사용할 수 있고 PyPI의 `importlib2`를 통해 파이썬 2에서 사용할 수 있는 `importlib`의 기능에 액세스해야 한다고 가정해 봅시다. 다음과 같이 예를 들어 `importlib.abc` 모듈을 액세스하는 코드를 작성하려고 할 수 있습니다:

```
import sys

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

이 코드는 문제점이 있는데, 파이썬 4가 나오면 어떻게 됩니까? 파이썬 3 대신 파이썬 2를 예외적인 사례로 취급하고 향후 파이썬 버전이 파이썬 2보다는 파이썬 3과 더 호환될 것이라고 가정하는 것이 좋습니다:

```
import sys

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

그러나 가장 좋은 해결책은 버전 감지를 않고 기능 감지에 의존하는 것입니다. 그러면 버전 감지가 잘못될 수 있는 잠재적인 문제를 피하고 미래 호환성을 유지할 수 있습니다:

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

2.6 호환성 회귀를 방지하십시오

일단 파이썬 3과 호환되도록 코드를 완전히 번역했으면, 코드가 회귀하고 파이썬 3에서 작동을 멈추는 것을 방지하고 싶을 것입니다. 이는 이 시점에 여러분이 실제로 파이썬 3에서 실행하는 것을 막는 의존성이 있는 경우에 특히 그렇습니다.

계속 호환되도록 하려면, 새로 만드는 모든 모듈의 맨 위에 최소한 다음 코드 블록이 있어야 합니다:

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

-3 플래그로 파이썬 2를 실행하여 코드가 실행되는 동안 발생하는 다양한 호환성 문제에 대해 경고를 받을 수도 있습니다. -Werror로 경고를 에러로 바꾸면 실수로 경고를 놓치지 않도록 할 수 있습니다.

Pylint 프로젝트와 그것의 --py3k 플래그를 사용하여 코드가 파이썬 3 호환성에서 벗어나기 시작할 때 경고를 받도록 코드를 검사할 수도 있습니다. 또한 이것은 호환성 회귀를 포착하기 위해 정기적으로 코드를 Modernize나 Futurize로 실행하지 않아도 되도록 합니다. 이것은 여러분이 파이썬 2.7과 파이썬 3.4 이상만 지원할 것을 요구합니다, 이것이 Pylint의 최소 파이썬 버전 지원이기 때문입니다.

2.7 전환을 막는 의존성을 확인하십시오

코드를 파이썬 3과 호환되게 만든 후에 의존성도 이식되었는지를 신경 쓰기 시작해야 합니다. caniusepython3 프로젝트는 어떤 프로젝트가 - 직접 또는 간접적으로 - 파이썬 3을 지원하는 것을 막는지 판단하는 데 도움을 주기 위해 만들어졌습니다. 명령 줄 도구뿐만 아니라 <https://caniusepython3.com> 에 웹 인터페이스가 있습니다.

이 프로젝트는 또한 테스트 스위트에 통합할 수 있는 코드를 제공해서 더는 파이썬 3 사용을 막는 의존성이 없을 때 테스트가 실패하도록 합니다. 이는 의존성을 수동으로 확인하지 않도록 하고 파이썬 3에서 실행할 수 있을 때 신속하게 알림을 받을 수 있도록 합니다.

2.8 파이썬 3 호환성을 나타내도록 setup.py 파일을 갱신하십시오.

일단 코드가 파이썬 3에서 작동하면, Programming Language :: Python :: 3을 포함하고 파이썬 2만 지원한다고 지정하지 않도록 setup.py의 분류를 갱신해야 합니다. 이것은 코드를 사용하는 사람에게 파이썬 2**와** 3을 지원한다는 것을 알려줄 것입니다. 이상적으로는 현재 지원하는 파이썬의 각 주/부 버전을 위한 분류를 추가하고 싶을 것입니다.

2.9 지속적인 통합을 통해 호환성을 유지하십시오

일단 파이썬 3에서 완전히 실행할 수 있다면 코드가 항상 파이썬 2와 3에서 작동하는지 확인하고 싶을 것입니다. 아마도 여러 파이썬 인터프리터에서 테스트를 실행하는 가장 좋은 도구는 `tox`입니다. 실수로 파이썬 2나 3 지원을 망가뜨리지 않도록 지속적인 통합 시스템과 `tox`를 통합할 수 있습니다.

바이트열을 문자열과 비교하거나 바이트열을 `int`와 비교할 때 (후자는 파이썬 3.5부터 사용 가능합니다) 예외를 일으키도록 파이썬 3 인터프리터에 `-bb` 플래그를 사용할 수도 있습니다. 기본적으로 다른 형 간의 비교는 단순히 `False`를 반환하지만, 텍스트/바이너리 데이터 처리의 분리나 바이트열에 대한 인텔싱에서 실수한다면 실수를 쉽게 찾을 수 없습니다. 이 플래그는 이러한 종류의 비교가 발생할 때 예외를 발생시켜, 실수를 훨씬 쉽게 추적할 수 있도록 합니다.

그리고 이것이 대부분입니다! 이 시점에서 여러분의 코드 기반은 파이썬 2와 3과 동시에 호환됩니다. 여러분의 테스트도 개발 중에 어떤 버전으로 테스트를 실행하는지와 관계없이 실수로 파이썬 2나 3 호환성을 망가뜨리지 않도록 설정되었습니다.

2.10 선택적 정적 형 검사 사용을 고려하십시오

여러분의 코드를 이식하도록 돕는 또 다른 방법은 코드에서 `mypy`나 `pytype`와 같은 정적 형 검사기를 사용하는 것입니다. 이 도구를 사용하면 코드가 파이썬 2에서 실행되는 것처럼 코드를 분석할 수 있으며, 그런 다음 코드가 파이썬 3에서 실행되는 것처럼 두 번째로 도구를 실행할 수 있습니다. 이처럼 정적 형 검사기를 두 번 실행하면 예를 들어 한 버전의 파이썬에서 다른 버전에 비해 바이너리 데이터형을 잘못 사용하고 있는지 발견할 수 있습니다. 코드에 선택적 형 힌트를 추가하면 API가 텍스트나 바이너리 데이터 중 어느 것을 사용하는지 명시적으로 명시할 수도 있어서 두 버전의 파이썬에서 예상대로 모든 것이 기능하도록 확인하는 데 도움을 줍니다.