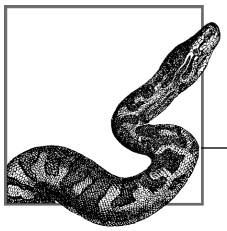

6
Модули

Типичная программа на Python состоит из нескольких файлов, в которых содержится исходный код. Каждый такой файл — это *модуль*, группирующий код и данные для повторного использования. Как правило, модули не зависят друг от друга и потому могут повторно использоваться другими программами. Иногда, для того чтобы было легче управлять многочисленными модулями, родственные модули группируют в *пакет* — иерархическую древовидную структуру.

Модуль явно устанавливает зависимости от других модулей посредством инструкций `import` и `from`. В некоторых других языках программирования для организации скрытых каналов связи между модулями используются глобальные переменные. В Python глобальные переменные не являются глобальными для всех модулей и выступают в качестве атрибутов одиночного объекта модуля. Таким образом, связь между модулями в Python всегда устанавливается явно и требует поддержки.

Кроме того, Python предоставляет *модули расширения* — это модули, написанные на других языках программирования, таких как C, C++, Java или C#, но предназначенные для использования в Python. Для кода Python, импортирующего модуль, не имеет значения, является ли данный модуль расширением или собственно модулем Python. Вы всегда можете начать с написания кода модуля на языке Python. Если впоследствии вам понадобится повысить быстродействие программы, вы сможете выполнить рефакторинг модулей и переписать некоторые их части с привлечением низкоуровневых языков, не внося никаких изменений в клиентский код, который использует эти модули. О том, как писать расширения на языках C и Cython, рассказано в главе 24.

В этой главе рассмотрены вопросы, относящиеся к созданию и загрузке модулей. Мы также обсудим группирование модулей в пакеты с помощью служебных утилит Python (`distutils` и `setuptools`), предназначенных для установки распространяемых пакетов и подготовки пакетов к распространению; эта тема более подробно раскрыта в главе 25. Данная глава завершается обсуждением наиболее оптимальных способов управления окружением Python.

Объекты модулей

Модуль — это объект Python с произвольно именованными атрибутами, которые можно связывать и на которые можно ссылаться. Обычно код модуля, носящего имя *name*, хранится в файле *name.py* (раздел "Загрузка модуля").

В Python модули являются объектами (значениями), с которыми можно обращаться как с любыми другими объектами. Таким образом, модули можно передавать в качестве аргументов функциям при их вызове. Точно так же функция может возвращать модуль в качестве результата вызова. Как и любой другой объект, модуль можно связать с переменной, элементом контейнера или атрибутом объекта. Модули могут быть ключами или значениями словаря, а также элементами множества. Например, в словаре `sys.modules` (раздел "Загрузка модуля") объекты модулей хранятся в качестве его значений. Тот факт, что модули можно рассматривать как любые другие значения в Python, часто формулируют в виде утверждения о том, что модули являются объектами *первого класса*.

Инструкция `import`

Чтобы использовать любой исходный файл Python в качестве модуля, другой исходный файл должен импортировать его с помощью инструкции `import`, которая имеет следующий синтаксис:

```
import имя_модуля [as имя_переменной][,...]
```

Вслед за ключевым словом `import` указываются спецификаторы одного или нескольких модулей, разделенные запятыми. В простейшем и наиболее распространенном случае спецификатор *имя_модуля* — это идентификатор, т.е. имя переменной, которую Python связывает с объектом модуля, имеющим то же имя, по завершении выполнения инструкции `import`. В этом случае Python ищет модуль с данным именем. Например, в случае следующей инструкции Python пытается найти модуль с именем `mymodule` и связать переменную `mymodule` с объектом данного модуля в текущей области видимости.

```
import mymodule
```

Спецификатор *имя_модуля* также может быть последовательностью идентификаторов, разделенных точками (`.`). Это позволяет ссылаться на модуль, хранящийся в пакете (раздел "Пакеты").

Если спецификатор модуля включает квалификатор `as`, то Python ищет модуль с именем *имя_модуля* и связывает объект модуля с переменной *имя_переменной*. Например, в случае инструкции

```
import mymodule as alias
```

Python ищет модуль с именем `mymodule` и связывает объект модуля с переменной `alias` в текущей области видимости. Переменная *имя_переменной* всегда должна быть простым идентификатором.

Тело модуля

Тело модуля — это последовательность инструкций, содержащихся в исходном файле модуля. Для указания того, что данный файл является модулем, не требуется использовать какой-либо специальный синтаксис: любой допустимый исходный файл Python может быть использован в качестве модуля. Инструкции, образующие тело модуля, немедленно выполняются, когда модуль впервые импортируется в выполняющуюся программу. В процессе выполнения тела модуля объект модуля уже создан и с ним уже связана соответствующая запись в словаре `sys.modules`. По мере выполнения тела модуля пространство имен (глобальное) модуля постепенно заполняется.

Атрибуты объектов модулей

Инструкция `import` создает новое пространство имен, содержащее все атрибуты данного модуля. Для доступа к атрибутам в этом пространстве имен следует использовать имя или псевдоним модуля в качестве префикса:

```
import mymodule
a = mymodule.f()
```

или

```
import mymodule as alias
a = alias.f()
```

Обычно атрибуты объекта модуля связываются с помощью инструкций в теле модуля. Если инструкция в теле модуля связывает (глобальную) переменную, то связываемый объект является атрибутом модуля.

Тело модуля существует для связывания атрибутов модуля

Обычно назначением тела модуля является создание атрибутов данного модуля: инструкции `def` создают и связывают функции, инструкции `class` создают и связывают классы, а инструкции присваивания могут связывать атрибуты любого типа. Ради ясности и чистоты кода следует избегать делать что-либо другое на верхнем логическом уровне в теле кода, кроме связывания атрибутов модуля.

Атрибуты модуля можно также связывать вне тела (например, в других модулях), обычно с использованием синтаксиса ссылок на атрибут вида *`M.имя`* (где *`M`* — любое выражение, значением которого является модуль, а идентификатор *`имя`* — имя атрибута). Однако для ясности обычно лучше все же связывать атрибуты модуля в теле собственного модуля.

Инструкция `import` устанавливает некоторые атрибуты модуля сразу после создания объекта модуля и до выполнения тела данного модуля. Атрибут `__dict__` — это объект словаря, который используется модулем в качестве пространства имен для своих атрибутов. В отличие от других атрибутов модуля, словарь `__dict__` не доступен коду внутри модуля в качестве глобальной переменной. Все остальные атрибуты в модуле являются элементами словаря `__dict__` и доступны коду внутри модуля в качестве глобальных переменных. Атрибут `__name__` — это имя модуля; атрибут `__file__` — имя файла, из которого загружен модуль. В других атрибутах, имена которых начинаются и заканчиваются двойными символами подчеркивания, хранятся другие метаданные модуля (в версии v3 количество таких атрибутов возросло во всех модулях).

Для любого объекта модуля *M*, любого объекта *x* и любого идентификатора строки *S* (за исключением `__dict__`) связывание *M*`.`*S* `=` *x* эквивалентно связыванию *M*`.__dict__['`*S*`'] =` *x*. Ссылка на атрибут вида *M*`.`*S* также в основном эквивалентна ссылке *M*`.__dict__['`*S*`']`. Единственным отличием является то, что в тех случаях, когда *S* не является ключом в словаре *M*`.__dict__`, ссылка *M*`.__dict__['`*S*`']` возбуждает исключение `KeyError`, тогда как ссылка *M*`.`*S* — исключение `AttributeError`. Кроме того, атрибуты модуля доступны во всем коде модуля в качестве глобальных переменных. Иными словами, использование *S* в качестве глобальной переменной в теле модуля эквивалентно использованию *M*`.`*S* (т.е. *M*`.__dict__['`*S*`']`) как в отношении связывания, так и ссылки (однако, если *S* не является ключом в словаре *M*`.__dict__`, то ссылка на *S* как на глобальную переменную возбуждает исключение `NameError`).

Встроенные объекты Python

Python предоставляет целый ряд встроенных объектов (рассматриваются в главе 7). Все встроенные объекты являются атрибутами модуля `builtins` (в версии v2 этот модуль называется `__builtin__`). Когда Python загружает какой-либо модуль, этот модуль автоматически получает дополнительный атрибут `__builtins__`, ссылающийся либо на модуль `builtins` (в версии v2 — `__builtin__`), либо на его словарь. Python может выбирать, какой из этих вариантов использовать, поэтому полагаться на данный атрибут не стоит. Если вам действительно необходимо получить непосредственный доступ к модулю `builtins` (а необходимость в этом может возникать лишь в редких случаях), то используйте инструкцию `import builtins` (в версии v2 — инструкцию `import __builtin__ as builtins`). В случае обращения к переменной, которую не удается найти ни в локальном, ни в глобальном пространстве имен текущего модуля, Python ищет этот идентификатор в словаре `__builtins__` текущего модуля, прежде чем возбудить исключение `NameError`.

Процедура поиска — единственный механизм, который Python предоставляет вам для того, чтобы ваш код мог получить доступ к встроенным объектам. Имена встроенных объектов не зарезервированы и не закодированы жестко в самом Python. Описанный механизм доступа прост и задокументирован, поэтому ваш код

может использовать его непосредственно (однако не переборщите, иначе от этого пострадают простота и ясность вашей программы). Поскольку Python обращается к встроенным объектам лишь в тех случаях, когда не может разрешить имя в локальной или глобальной области видимости, обычно достаточно определить подмену в одном из этих пространств имен. При этом вы можете добавлять собственные встроенные объекты или подменять обычные встроенные функции собственными, и тогда добавленные или подмененные объекты будут видны всем модулям. В следующем иллюстративном примере, ориентированном на версию v3, показано, как упаковать встроенную функцию в собственную функцию, чтобы разрешить функции `abs()` принимать строковый аргумент (и возвращать измененную строку).

```
# abs принимает числовой аргумент; позволим этой функции
# принимать также строку
import builtins
_abs = builtins.abs      # сохранить исходную встроенную функцию
def abs(str_or_num):
    if isinstance(str_or_num, str):    # если arg - строка
        return ''.join(sorted(set(str_or_num)))  # получить взамен
    return _abs(str_or_num)  # вызвать реальную встроенную функцию
builtins.abs = abs    # переопределить встроенную функцию
                      # с помощью функции-обертки
```

Единственное, что необходимо изменить в этом примере для того, чтобы он мог работать в версии v2, — это заменить инструкцию `import builtins` инструкцией `import __builtin__ as builtins`.

Строки документирования модулей

Если первой строкой в теле модуля является строковый литерал, то Python связывает эту строку с атрибутом строки документирования модуля `__doc__`. Строки документирования рассматривались в разделе "Другие атрибуты объектов функций" главы 3.

Закрытые переменные модулей

Ни одна переменная модуля не является истинно закрытой (частной). Однако в соответствии с принятым соглашением любой идентификатор, имя которого начинается с одиночного символа подчеркивания (_), такой как `_secret`, считается закрытым. Другими словами, ведущий символ подчеркивания сообщает программистам клиентского кода о том, что они не должны пытаться получать непосредственный доступ к такому идентификатору.

Среды разработки и другие инструменты следуют этому соглашению для того, чтобы отличать общедоступные атрибуты модуля (т.е. часть интерфейса модуля) от закрытых (т.е. тех, которые должны использоваться только в модуле).

Придерживайтесь соглашения об использовании ведущего символа подчеркивания в именах закрытых атрибутов

Очень важно, чтобы вы соблюдали соглашение об использовании ведущего символа подчеркивания в именах переменных, имитирующих закрытые переменные, особенно если вы пишете клиентский код, который использует модули, написанные другими людьми. Избегайте использования любых переменных, имена которых начинаются с символа _. Можно предположить, что будущие выпуски этих модулей будут поддерживать их общедоступные интерфейсы, но частные детали их реализации, скорее всего, изменятся, и закрытые атрибуты предназначены для реализации именно таких деталей.

Инструкция `from`

Инструкция `from` позволяет импортировать конкретные атрибуты из модуля в текущее пространство имен. Инструкция `from` имеет два варианта синтаксиса:

```
from имя_модуля import имя_атрибута [as имя_переменной][,...]
from имя_модуля import *
```

Инструкция `from` определяет имя модуля, за которым следует один или несколько спецификаторов, разделенных запятыми. В простейшем и наиболее распространенном случае спецификатор атрибута — это идентификатор *имя_атрибута*, т.е. имя переменной, которую Python связывает с одноименным атрибутом, принадлежащим модулю *имя_модуля*, например:

```
from mymodule import f
```

Кроме того, *имя_модуля* может также быть последовательностью идентификаторов, разделенных точками (`.`), что позволяет ссылаться на модуль, хранящийся в пакете (раздел “Пакеты”).

Если спецификатор атрибута включает квалификатор `as`, то Python получает из модуля значение атрибута *имя_атрибута* и связывает его с переменной *имя_переменной*, например:

```
from mymodule import f as foo
```

Учтите, что *имя_атрибута* и *имя_переменной* всегда должны быть простыми идентификаторами.

При желании вы можете заключить в круглые скобки всю группу спецификаторов атрибутов, следующих за ключевым словом `import` в инструкции `from`. Иногда это может быть полезным, если у вас есть несколько спецификаторов атрибутов и вы хотите разбить одну логическую строку инструкции `from` на несколько логических строк более элегантным способом, чем с помощью символов обратной косой черты (\).

```
from some_module_with_a_long_name import (
  another_name, and_another as x, one_more, and_yet_another as y)
```

Инструкция `from ... import *`

Код, содержащийся непосредственно в теле модуля (а не в теле функции или класса), может использовать символ "звездочка" (`*`) в инструкции `from`:

```
from имя_модуля import *
```

Символ `*` запрашивает связывание всех атрибутов модуля *имя_модуля* в качестве глобальных переменных. Если модуль *имя_модуля* имеет атрибут `__all__`, то значением этого атрибута является список имен атрибутов, связанных этим типом инструкции `from`. В противном случае этот тип инструкции `from` связывает все атрибуты модуля *имя_модуля*, за исключением тех, имена которых начинаются с символов подчеркивания.

Остерегайтесь использовать инструкцию `from M import *` в своем коде

Поскольку инструкция `from M import *` может связывать произвольный набор глобальных переменных, в ней заложен риск нежелательных побочных эффектов, таких как сокрытие встроенных функций и изменение связывания нужных вам переменных. Если вы используете инструкцию `from` в такой форме, то делать это следует очень расчетливо и лишь в отношении тех модулей, в документации которых явно указано, что они поддерживают подобный способ их использования. Вероятно, лучше всего воздержаться от использования указанной формы инструкции в коде и рассматривать ее лишь в качестве вспомогательного средства, которое в отдельных случаях удобно использовать в интерактивных сеансах работы с Python.

Сравнение инструкций `from` и `import`

Чаще всего инструкция `import` является более предпочтительной по сравнению с инструкцией `from`. Рассматривайте инструкцию `from`, особенно в форме `from M import *`, как удобное средство, предназначенное лишь для эпизодического использования в интерактивных сеансах работы с Python. Если вы всегда будете получать доступ к модулю `M` с помощью инструкции `import M` и обращаться к его атрибутам с помощью явного синтаксиса `M.A`, то ваш код лишь немного потеряет в лаконичности, но при этом значительно выиграет в ясности и читаемости. К числу случаев, в которых использование инструкции может быть оправданным, относится импорт отдельных модулей пакета (раздел "Пакеты"). Но в большинстве случаев целесообразнее использовать инструкцию `import`, а не `from`.

Загрузка модуля

Операции по загрузке модуля реализованы во встроенной функции `__import__` и в процессе своего выполнения используют атрибуты встроенного модуля `sys`

(раздел “Модуль sys” в главе 7). Ваш код может вызывать функцию __import__ непосредственно, но в современном Python поступать так категорически не рекомендуется; вместо этого следует выполнить инструкцию import importlib и вызвать функцию importlib.import_module, передав ей строку с именем модуля в качестве единственного аргумента. Функция import_module возвращает объект модуля или, если операцию импорта не удается выполнить, возбуждает исключение ImportError. В то же время имеет смысл глубже разобраться в семантике функции __import__, поскольку как функция import_module, так и инструкция import зависят от нее.

Чтобы импортировать модуль *M*, функция __import__ сначала просматривает словарь sys.modules, используя строку *M* в качестве ключа. Если ключ *M* имеется в словаре, функция __import__ возвращает соответствующее значение в качестве запрошенного объекта модуля. В противном случае функция __import__ связывает значение sys.modules[*M*] с новым пустым объектом модуля, атрибут __name__ которого содержит строку *M*, а затем ищет подходящий способ инициализации (загрузки) модуля (раздел “Поиск модуля в файловой системе”).

Благодаря этому механизму относительно медленная операция загрузки выполняется лишь тогда, когда данный модуль впервые импортируется в текущую выполняющуюся программу. В случае повторного импортирования модуля он не загружается заново, поскольку функция __import__ быстро находит и возвращает из словаря sys.modules запись, соответствующую данному модулю. Таким образом, все операции импортирования модуля, кроме первой, выполняются очень быстро и сводятся лишь к просмотру словаря. (Относительно возможности *принудительной* перезагрузки модуля см. в разделе “Перезагрузка модулей”.)

Встроенные модули

Когда загружается модуль, функция __import__ сначала проверяет, является ли он встроенным. Список всех встроенных модулей содержится в кортеже sys.builtin_module_names, но повторное связывание этого кортежа не влияет на загрузку модулей. Когда Python загружает встроенный модуль, он вызывает его функцию инициализации, как и при загрузке любого другого расширения. Кроме того, поиск модулей осуществляется в расположениях, зависящих от платформы, таких как реестр Windows.

Поиск модуля в файловой системе

Если модуль *M* не является встроенным, то функция __import__ ищет файл, содержащий его код, в файловой системе. Функция __import__ поочередно проверяет строки, являющиеся элементами списка sys.path. Каждый элемент списка представляет путь к каталогу или к архивному файлу, упакованному в популярном ZIP-формате. Список sys.path инициализируется при запуске программы с использованием переменной среды PYTHONPATH (рассматривалась в разделе “Переменные среды” главы 2), если она

имеется. Первым в `sys.path` всегда указывается каталог, из которого загружена основная программа. Пустая строка в `sys.path` соответствует текущему каталогу.

Ваш код может изменить или повторно связать `sys.path`, и такие изменения влияют на то, в каких каталогах и ZIP-архивах функция `__import__` осуществляет поиск загружаемых модулей. Однако эти изменения не влияют на те модули, которые к моменту внесения изменений в `sys.path` уже были загружены (а значит, уже записаны в словарь `sys.modules`).

Если при запуске программы в каталоге *PYTHONHOME* обнаружен файл с расширением *.pth*, то содержимое этого файла, каждый элемент которого должен располагаться в отдельной строке, добавляется в список путей, хранящихся в `sys.path`. Файлы с расширением *.pth* могут содержать пустые строки и комментарии, начинающиеся с символа `#`; Python игнорирует такие строки. В этих файлах также могут содержаться инструкции `import` (которые выполняются прежде, чем начнет выполняться ваша программа), но никакие другие инструкции не допускаются.

В процессе просмотра каждого из каталогов и ZIP-архивов, упомянутых в списке `sys.path`, Python выполняет поиск следующих файлов модуля *`M`*, перечисленных ниже в том порядке, в каком осуществляется поиск.

Файлы с расширениями *.pyd* и *.dll* (Windows) или *.so* (большинство Unix-подобных платформ), соответствующие модулям расширения Python. (В некоторых диалектах Unix используются другие расширения, например *.sl* в HP-UX.) На большинстве платформ расширения не могут загружаться из ZIP-архива — годятся только модули в виде файлов с исходным кодом или скомпилированным байт-кодом.

Файлы с расширением *.py*, соответствующие исходному коду модулей Python.

Файлы с расширениями *.pyc* (или *.pyo* в версии v2, если Python выполняется с опцией `-O`), соответствующие скомпилированным в байт-код модулям Python.

Если обнаружен файл с расширением *.py*, то (только в версии v3) Python ищет также каталог *`__pycache__`*. Если такой каталог найден, то Python ищет в нем файлы с расширением *.<tag>.pyc*, где *<tag>* — строка, специфическая для версии Python, которая выполняет поиск модуля.

Последний путь, который Python использует в процессе поиска файла модуля *`M`* — это *__init__.py*, т.е. ищется файл с именем *__init__.py* в каталоге с именем *`M`* (раздел "Пакеты").

Найдя исходный файл *M.py*, Python (v3) компилирует его в файл *M.<tag>.pyc*, кроме тех случаев, когда был найден скомпилированный файл, созданный позже файла *M.py*, причем компиляция выполнялась в той же версии Python, в которой был создан исходный файл. Если *M.py* компилировался из каталога, разрешающего запись, то Python создает по необходимости каталог `__pycache__` и сохраняет скомпилированный файл в этом подкаталоге файловой системы, чтобы впоследствии не компилировать его заново. (В версии v2 файл *M.py* компилируется в файл *M.pyc* или *M.pyo*, который Python сохраняет в том же каталоге, в котором находится файл *M.py*.) Если найденный скомпилированный файл создан позже исходного файла (это

определяется на основании внутренней временной метки файла, содержащего байт-код, а не даты, записанной в файловой системе), то Python повторно не компилирует модуль.

Получив файл с байт-кодом, созданный посредством компиляции или прочитанный из файловой системы, Python выполняет тело модуля для инициализации объекта модуля. Если модуль является расширением, Python вызывает функцию инициализации модуля.

Основная программа

Обычно выполнение приложения Python начинается со сценария верхнего уровня, также называемого *основной программой* (см. раздел "Программа `python`" в главе 2). Основная программа выполняется подобно любому другому загружаемому модулю, за исключением того, что Python сохраняет скомпилированный байт-код в памяти, но не сохраняет на диске. Модуль основной программы всегда носит одно и то же имя — `'__main__'`, хранящееся как в глобальной переменной `__name__` (атрибут модуля), так и в виде ключа в словаре `sys.modules`.

Не импортируйте *.py*-файл, который используете в качестве основной программы

Не следует импортировать тот же *.py*-файл, который является основной программой. Если вы сделаете это, то модуль загрузится вновь и его тело выполнится еще раз с самого начала в отдельном объекте модуля с другим значением атрибута `__name__`.

Код модуля Python может протестировать, является ли используемый модуль основной программой, проверив, имеет ли глобальная переменная `__name__` значение `'__main__'`. Показанную ниже идиому часто используют для защиты кода, чтобы он выполнялся только в том случае, если модуль запускается в качестве основной программы.

```
if __name__ == '__main__':
```

Если модуль предназначен только для импортирования, он должен нормально проходить блочные тесты, когда выполняется в качестве основной программы (раздел "Модульное и системное тестирование" в главы 16).

Перезагрузка модулей

Python загружает модуль только тогда, когда он в первый раз импортируется во время выполнения программы. Если разработка ведется в интерактивном режиме, то необходимо явно перезагружать модули после каждого их изменения (некоторые среды разработки обеспечивают автоматическую перезагрузку модулей).

Чтобы перезагрузить модуль в версии v3, передайте объект модуля (а не его имя) в качестве единственного аргумента функции `reload` из модуля `importlib` (в версии v2 вызовите вместо этого встроенную функцию `reload`, которая дает тот же результат). Вызов `importlib.reload(`*`M`*`)` обеспечивает гарантированное использование перезагруженной версии *`M`* клиентским кодом, полагающимся на инструкцию `import` *`M`* и получающим доступ к атрибутам с помощью синтаксиса *`M.A`*. Однако вызов `importlib.reload(`*`M`*`)` не оказывает влияния на другие существующие ссылки, связанные с предыдущими значениями атрибутов *`M`* (например, с помощью инструкции `from`). Другими словами, вызов функции `reload` не влияет на ранее связанные переменные, связывание которых остается прежним. Неспособность функции `reload` повторно связывать такие переменные служит дополнительным поводом к использованию инструкции `import` вместо инструкции `from`.

Функция `reload` не является рекурсивной: перезагрузка модуля *`M`* не означает, что при этом перезагружаются также другие модули, импортируемые модулем *`M`*. Вы должны перезагружать каждый модуль по отдельности путем явных вызовов функции `reload`.

Циклический импорт

Python позволяет определять циклические операции импорта. Например, у вас может быть модуль *a.py*, содержащий инструкцию `import` *`b`*, в то время как модуль *b.py* содержит инструкцию `import` *`a`*.

Если вы все же используете циклический импорт по каким-либо причинам, то должны хорошо понимать, как он работает, чтобы избежать ошибок в своем коде.

Избегайте циклического импорта

На практике почти всегда лучше избегать операций циклического импорта, поскольку циклические зависимости хрупкие и ими трудно управлять.

Предположим, что основной сценарий выполняет инструкцию `import a`. В соответствии с предыдущим обсуждением эта инструкция создает новый пустой объект модуля в виде значения словаря `sys.modules['a']`, после чего начинает выполняться тело модуля `a`. Выполнение в теле модуля `a` инструкции `import b` создает новый пустой объект модуля в виде значения словаря `sys.modules['b']`, после чего начинает выполняться тело модуля `b`. Выполнение тела модуля `a` не может быть продолжено до тех пор, пока не закончится выполнение тела модуля `b`.

В процессе выполнения модулем `b` инструкции `import a` она выясняет, что значение `sys.modules['a']` уже связано, и поэтому связывает глобальную переменную `a` в модуле `b` с объектом модуля `a`. Поскольку выполнение тела модуля `a` в это время блокировано, то обычно на данном этапе модуль `a` заполняется лишь частично. Тогда

любая попытка кода модуля b получить немедленный доступ к атрибуту модуля a, который к этому времени еще не был связан, приведет к возникновению ошибки.

Если в силу каких-то причин вы все же хотите использовать циклический импорт, то вам следует тщательно продумать, в какой очередности каждый из модулей должен связывать свои глобальные переменные, импортировать другие модули и получать доступ к атрибутам других модулей. Вы можете получить больший контроль над последовательностью развития событий, группируя свои инструкции в функции и вызывая эти функции в контролируемой очередности, а не просто полагаться на последовательность выполнения инструкций верхнего уровня в телах модулей. Обычно проще отказаться от циклического импорта, чем пытаться гарантировать совершенно безопасный порядок выполнения операций в случае циклического импорта.

Изменение записей в словаре `sys.modules`

Функция `__import__` никогда не связывает в качестве значения в словаре `sys.modules` любые другие объекты, отличные от объектов модулей. Однако, если функции `__import__` удается найти в словаре `sys.modules` уже имеющуюся запись, она возвращает соответствующее значение независимо от его типа. Инструкции `import` и `from` внутренне полагаются на результат, возвращаемый функцией `__import__`, и поэтому все может закончиться тем, что они будут использовать объекты, не являющиеся модулями. Это позволяет устанавливать экземпляры классов в качестве записей в словаре `sys.modules`, что предоставляет возможность воспользоваться такими, например, средствами, как специальные методы `__getattr__` и `__setattr__` (см. раздел "Универсальные специальные методы" в главе 4). Этот редко используемый прием, не относящийся к числу простых, позволяет импортировать другие объекты, как если бы они являлись модулями, и вычислять их атрибуты на лету. Рассмотрим соответствующий иллюстративный пример.

```
class TT(object):
    def __getattr__(self, name): return 23
import sys
sys.modules[__name__] = TT()
```

Когда этот код импортируется первый раз в качестве модуля, он переопределяет относящуюся к модулю запись в словаре `sys.modules` экземпляром класса *`TT`*. При попытке получения значения атрибута с произвольным именем для экземпляра этого класса всегда будет возвращаться значение `23`.

Пользовательские операции импорта

Суть одного из усложненных и редко используемых видов функциональности, предлагаемых Python, заключается в возможности изменения семантики части или всех операций `import` и `from`.

Повторное связывание атрибута `__import__`

Вы можете изменить связывание атрибута `__import__` модуля `builtin`, повторно связав его с собственной функцией-импортером, например, используя общую методику обертывания встроенных функций (см. раздел "Встроенные объекты Python"). Подобное повторное связывание оказывает влияние на все инструкции `import` и `from`, которые выполняются после него, и, таким образом, может иметь нежелательные глобальные эффекты. Пользовательская функция-импортер, созданная путем повторного связывания атрибута `__import__`, должна реализовывать те же интерфейс и семантику, что и встроенная функция `__import__`, и, в частности, нести ответственность за корректное использование словаря `sys.modules`.

Избегайте изменения связывания встроенного атрибута `__import__`

Несмотря на то что повторное связывание атрибута `__import__` поначалу может показаться привлекательным подходом, в большинстве случаев, когда требуются пользовательские функции-импортеры, лучше всего реализовать их посредством функций — перехватчиков операции импорта (рассмотрены ниже).

Перехватчики импорта

Python предлагает расширенную поддержку избирательного изменения деталей выполнения операции импорта. Пользовательские функции-импортеры относятся к числу более сложных и редко используемых приемов, однако некоторые приложения могут нуждаться в них для таких, например, целей, как импортирование кода из архивов, отличных от ZIP-файлов, а также из баз данных, сетевых серверов и т.п.

Наиболее подходящим способом удовлетворения подобных завышенных запросов является запись вызываемых объектов-импортеров в качестве элементов атрибутов `meta_path` и/или `path_hooks` модуля `sys`, о чем подробно говорится в документе **PEP 302** (`https://www.python.org/dev/peps/pep-0302/`), а также, применительно к версии v3, в документе **PEP 451** (`https://www.python.org/dev/peps/pep-0451/`). Именно так Python перехватывает обращение к модулю `zipimport` стандартной библиотеки для обеспечения беспрепятственного импорта модулей из ZIP-файлов, о чем упоминалось ранее. Если вы собираетесь интенсивно использовать перехватчики `sys.path_hooks` и родственные им, то без тщательного изучения документов **PEP 302** и **PEP 451** вам не обойтись. Однако, для того чтобы вы могли получить некоторое представление об этих возможностях, если потребность в них у вас когда-либо возникнет, ниже приведен небольшой иллюстративный пример.

Предположим, в процессе разработки первого наброска некоторой программы вам было бы удобно использовать инструкции `import` в отношении модулей, которые еще не написаны, получая при том всего лишь сообщения (и пустые модули). Вы можете получить такую функциональность (оставляя в стороне сложности,

связанные с пакетами, и работая только с простыми модулями) путем создания пользовательского модуля импортера, как показано ниже.

```
import sys, types

class ImporterAndLoader(object):
    '''Импортер и загрузчик часто совмещаются в одном классе'''
    fake_path = '!dummy!'
    def __init__(self, path):
        # Обрабатывать только собственный фиктивный маркер пути
        if path != self.fake_path: raise ImportError
    def find_module(self, fullname):
        # Даже не пытаться обрабатывать любое полное имя модуля
        if '.' in fullname: return None
        return self
    def create_module(self, spec):
        # Выполняется только в v3: создает модуль
        # принятым по умолчанию способом
        return None
    def exec_module(self, mod):
        # Выполняется только в v3: заполняет уже
        # инициализированный модуль;
        # в этом примере просто выводится сообщение
        print('ПРИМЕЧАНИЕ: module {!r} еще не написан'.format(mod))
    def load_module(self, fullname):
        # Выполняется только в v2: создает и заполняет модуль
        if fullname in sys.modules: return sys.modules[fullname]
        # В этом примере просто выводится сообщение
        print('ПРИМЕЧАНИЕ: module {!r} еще не написан'.format(fullname))
        # Создать новый пустой модуль, занести его в sys.modules
        mod = sys.modules[fullname] = types.ModuleType(fullname)
        # Минимальная инициализация нового модуля и его возврат
        mod.__file__ = 'dummy_{}'.format(fullname)
        mod.__loader__ = self
        return mod
# Добавить класс в path_hooks, а фиктивный маркер его пути - в path
sys.path_hooks.append(ImporterAndLoader)
sys.path.append(ImporterAndLoader.fake_path)

if __name__ == '__main__':    # самотестирование при запуске в
                              # качестве основного сценария
    import missing_module     # импортировать простой отсутствующий
                              # модуль
    print(missing_module)     # должно успешно выполниться
    print(sys.modules.get('missing_module'))  # также должно успешно
                                              # выполниться
```

В версии v2 необходимо реализовать метод `load_module`, который помимо всего прочего должен выполнить некоторые стереотипные задачи, такие как обработка словаря `sys.modules` и установка таких атрибутов, имена которых начинаются и заканчиваются двумя символами подчеркивания, как `__file__`, для объекта модуля. В версии v3 система сама выполняет эти стереотипные задачи, поэтому нам достаточно написать тривиальные версии методов `create_module` (который в данном случае всего лишь возвращает значение `None`, запрашивая систему создать объект модуля способом, используемым по умолчанию) и `exec_module` (получает объект модуля с уже инициализированными атрибутами, имена которых содержат двойные символы подчеркивания, и надлежащим образом заполняет его значениями, как обычно).

В версии v3 мы могли бы воспользоваться новым мощным понятием *спецификации модуля*. Однако это потребовало бы использования модуля `importlib` стандартной библиотеки, который в большинстве случаев отсутствует в версии v2; более того, в столь небольшом примере эти мощные возможности нам просто не нужны. Вместо этого мы реализовали метод `find_module`, который в любом случае необходим для версии v2, и, несмотря на то что в настоящее время он является устаревшим, работает и в версии v3 в целях обеспечения обратной совместимости.

Пакеты

Пакет — это модуль, содержащий другие модули. Некоторые или все модули пакета могут быть подпакетами (подчиненными пакетами), что приводит к древовидной иерархической структуре пакета. Пакет *`P`* создается как одноименный подкаталог некоторого каталога, указанного в `sys.path`. Кроме того, пакеты могут существовать в виде ZIP-файлов. В этом разделе предполагается, что пакет находится в файловой системе, однако то же самое относится и к пакетам в виде ZIP-файлов, поскольку они также используют внутреннюю иерархическую структуру ZIP-файла.

Тело модуля *`P`* содержится в файле *P/__init__.py*. Этот файл должен существовать (это не относится к пакетам пространств имен в версии v3, рассмотренным в разделе "Пакеты пространств имен (только в версии v3)"), даже если он пустой (что означает пустое тело модуля), тем самым сообщая Python, что *`P`* действительно является пакетом. Тело модуля пакета загружается, когда вы в первый раз импортируете пакет (или любой из модулей пакета), и ведет себя, как любой другой модуль Python. Другие *.py*-файлы в каталоге *P* — это модули пакета *`P`*. Подкаталоги *P*, содержащие файлы *__init__.py*, являются *подпакетами P*. Вложение пакетов может распространяться на любую глубину.

Модуль *`M`*, содержащийся в пакете *`P`*, можно импортировать как *`P.M`*. Дополнительные точки обеспечивают навигацию по иерархической структуре пакета. (Тело модуля пакета всегда загружается до загрузки любого из содержащихся в пакете модулей.) Если вы используете синтаксис `import P.M`, то переменная *P* связывается с объектом модуля пакета *`P`*, а атрибут *`M`* объекта *`P`* связывается с модулем *`P.M`*. Если

вы используете синтаксис `import P.M as V`, то с модулем `P.M` связывается непосредственно переменная `V`.

Использование инструкции `from P import M` для импортирования конкретного модуля `M` из пакета `P` абсолютно допустимо и в действительности настоятельно рекомендуется. Инструкция `from P import M as V` также вполне допустима и полностью эквивалентна инструкции `import P.M as V`.

Если содержащийся в пакете `P` модуль `M` выполняет инструкцию `import X`, то в версии v2 Python по умолчанию осуществляет поиск `X` в `M`, прежде чем искать его в `sys.path`. Однако в версии v3 это не так: чтобы избежать двусмысленности семантик, мы настоятельно рекомендуем использовать в версии v2 инструкцию `from __future__ import absolute_import`, чтобы сделать поведение версии v2 в этом отношении аналогичным поведению версии v3. Тогда модуль `M` содержащийся в пакете `P`, может явно импортировать модуль `X`, относящийся к тому же уровню (т.е. также содержащийся непосредственно в пакете `P`), с помощью инструкции `from . import X`.

Совместное использование объектов модулями одного пакета

Простейший и наиболее чистый способ совместного использования (разделения) объектов (например, функций или констант) модулями, содержащимися в пакете `P`, заключается в группировании разделяемых объектов в модуле, которому обычно присваивают имя *P/common.py*. Благодаря этому вы сможете использовать инструкцию `from . import common` в каждом из модулей пакета, которые нуждаются в доступе к общим объектам, а затем обращаться к этим объектам с помощью ссылок `common.f`, `common.K` и т.п.

Специальные атрибуты объектов пакетов

Атрибут `__file__` пакета `P` — это строка, представляющая путь к телу модуля `P`, т.е. путь к файлу *P/__init__.py*. Атрибут `__package__` пакета `P` — это имя пакета `P`.

Тело модуля пакета `P` — т.е. исходный код на языке Python, содержащийся в файле *P/__init__.py*, — может (как и любой другой модуль) определить глобальную переменную `__all__`, позволяющую контролировать, что будет происходить, если другой код на языке Python выполнит инструкцию `from P import *`. В частности, если переменная `__all__` не установлена, то инструкция `from P import *` импортирует не модули, содержащиеся в пакете `P`, а только имена, которые заданы в теле модуля `P` и не начинаются с символа подчеркивания (_). Как бы то ни было, использовать этот подход *не* рекомендуется.

Атрибут `__path__` пакета `P` — это список строк, представляющих пути к каталогам, из которых загружаются модули и подпакеты `P`. Первоначально Python устанавливает с помощью атрибута `__path__` список, включающий единственный элемент: путь к каталогу, содержащему файл *__init__.py*, который является телом модуля пакета. Ваш код может изменить этот список, чтобы повлиять на поиск модулей и подпакетов данного

пакета, осуществляемый впоследствии. Необходимость в применении такой методики возникает редко, но она может быть полезной, если вы хотите поместить модули пакета в несколько разрозненных каталогов. Только в версии v3: рассмотренные ниже пакеты пространств имен — обычный способ решения указанной задачи.

Пакеты пространств имен (только в версии v3)

Только в версии v3, если при выполнении инструкции `import foo` оказывается, что один или несколько каталогов, являющихся непосредственными подкаталогами элементов `sys.path`, называются *foo* и ни в одном из них не содержится файл *__init__.py*, то Python приходит к заключению, что *foo* — это *пакет пространства имен*. Как результат, Python создает (и присваивает элементу словаря `sys.modules['foo']`) объект пакета `foo` без атрибута `__file__`. Атрибут `foo.__path__` — это список всех каталогов, образующих данный пакет (и, как и в случае обычных пакетов, ваш код может вносить изменения в данный список). Следует отметить, что эта возможность относится к числу редко используемых.

Абсолютный и относительный импорт

Как отмечалось в разделе "Пакеты", обычно инструкция `import` осуществляет поиск целевого объекта в одном из расположений, указанных в `sys.path`, — поведение, описываемое термином *абсолютный импорт* (чтобы гарантировать такое же надежное поведение в версии v2, следует использовать инструкцию `from __future__ import absolute_import`). Кроме того, можно явно использовать *относительный импорт*, соответствующий импорту объекта из текущего пакета. При относительном импорте имена модулей или пакетов начинаются с одной или нескольких точек и доступны лишь в случае инструкции `from`. Инструкция `from . import X` ищет модуль или объект с именем *X* в текущем пакете; инструкция `from .X import y` ищет модуль или объект с именем *y* в модуле или подпакете *X*, находящемся в текущем пакете. Если ваш пакет имеет подпакеты, то их код может получать доступ к объектам, расположенным на более высоких уровнях иерархии пакета, путем использования нескольких точек в начале имени модуля или подпакета, помещаемого между ключевыми словами `from` и `import`. Каждая дополнительная точка соответствует переходу вверх на один уровень иерархии каталогов. Чрезмерное использование этой возможности вредит ясности кода, поэтому пользуйтесь ею с осторожностью и лишь тогда, когда это действительно необходимо.

Утилиты распространения (`distutils`) и установки (`setuptools`)

Модули, расширения и приложения Python можно упаковывать и распространять в нескольких формах.

Сжатые архивные файлы
: Обычно это файлы *.zip* или *.tar.gz* (также известны как файлы *.tgz*), причем обе эти формы переносимы, но существует также много других форм сжатия и архивирования деревьев файлов и каталогов.

Самораспаковывающиеся и самоустанавливающиеся исполняемые файлы
: Как правило, это файлы *.exe* для Windows.

Автономные, готовые к выполнению исполняемые файлы, не требующие установки
: Например, файлы *.exe* для Windows, ZIP-архивы с короткими префиксами сценариев для Unix, файлы *.app* для Mac и т.п.

Платформозависимые установочные пакеты
: Например, пакеты *.msi* для Windows, *.rpm* и *.srpm* для многих дистрибутивов Linux, *.deb* для Debian GNU/Linux и Ubuntu, *.pkg* для macOS.

Расширения Python `Wheels` (и `Eggs`)
: Популярные расширения от сторонних производителей (раздел "Архивные форматы *wheels* и *eggs*").

Чтобы установить пакет, распространяемый в виде самоустанавливающегося исполняемого файла или платформозависимого установщика, пользователю достаточно запустить установщик на выполнение. Способ запуска программы-установщика зависит от платформы, но не зависит от языка, на котором была написана программа. О создании таких самоустанавливающихся исполняемых файлов для различных платформ рассказано в главе 25.

Если же пакет распространяется в виде архивного файла или исполняемого файла, который распаковывается, но не устанавливается самостоятельно, то важно, чтобы пакет был написан на Python. В таком случае пользователь прежде всего должен распаковать файл в подходящий каталог, скажем, в каталог *C:\Temp\MyPack* на компьютере Windows или *~/MyPack* в Unix-подобной системе. Среди извлекаемых файлов обязательно должен быть файл с общепринятым именем *setup.py*, который использует средства Python, известные как *утилиты распространения* (пакет `distutils` стандартной библиотеки) или более популярный и мощный пакет `setuptools` от сторонних производителей (`https://pypi.python.org/pypi/setuptools`). В этом случае распространяемый пакет устанавливается так же просто, как и самоустанавливающийся исполняемый файл. Пользователь открывает окно командной строки и переходит в каталог с распакованным архивом. Затем необходимо выполнить примерно следующую команду:

```
C:\Temp\MyPack> python setup.py install
```

(В настоящее время более предпочтительный способ установки пакетов предлагает система управления пакетами `pip`, кратко рассмотренная в разделе "Окружения Python".) Сценарий *setup.py*, запускаемый этой командой `install`, устанавливает

пакет как часть установки Python данного пользователя в соответствии с опциями, определенными автором пакета в сценарии установки. Разумеется, пользователь должен иметь соответствующие разрешения для записи в каталоги установки Python, и в этом случае может потребоваться использование таких команд, позволяющих повышать привилегии, как *sudo*, но еще лучше использовать для установки пакетов виртуальное окружение (раздел "Окружения Python"). По умолчанию утилиты `distutils` и `setuptools` выводят информацию, когда пользователь запускает сценарий *setup.py*. Опция `---quiet`, указываемая непосредственно перед командой `install`, позволяет скрыть большинство деталей (при этом пользователь по-прежнему видит сообщения об ошибках, если таковые имеются). Следующая команда предоставляет подробную справочную информацию относительно пакета `distutils` или `setuptools`, в зависимости от того, какой из этих инструментов автор пакета использовал в своем файле *setup.py*:

```
C:\Temp\MyPack> python setup.py --help
```

Последние выпуски обеих версий, v2 и v3, поставляются с превосходным установщиком `pip` (рекурсивный акроним от "pip installs packages"), который детально документирован в онлайн-руководстве (`https://pip.pypa.io/en/stable/user_guide/`) и отличается простотой использования в большинстве случаев. Команда `pip install` *`пакет`* находит онлайн-версию указанного пакета (обычно в огромном репозитории PyPI [`https://pypi.python.org/pypi`], насчитывающем почти 100000 пакетов на момент выхода данной книги), загружает его и устанавливает для вас (в виртуальное окружение, если одно из них активно; см. раздел "Окружения Python"). Именно этот простой, но мощный подход использовался авторами данной книги для установки пакетов в более чем 90% случаев.

Даже если вы загрузили пакет локально (скажем, в каталог */tmp/mypack*) по каким-либо причинам (возможно, он отсутствует в PyPI, или вы пытаетесь экспериментировать с версией, которая там еще не сохранена), `pip` по-прежнему сможет установить его: для этого достаточно выполнить команду `pip install --no-index --findlinks=/tmp/mypack`, и `pip` сделает все остальное.

Архивные форматы *wheels* и *eggs*

Формат *wheels* (как и его предшественник *eggs*, все еще поддерживаемый, но не рекомендуемый для будущих разработок) — это архивный формат Python, включающий структурированные метаданные, а также код на Python. Оба формата, особенно *wheels*, предлагают отличные возможности для упаковки и распространения пакетов Python, и средство `setuptools` (вместе с расширением `wheel`, которое, конечно же, легко установить с помощью команды `pip install wheel`) без проблем работает совместно с ними. Более подробно об этих форматах можно прочитать на сайте `pythonwheels.com`, а также в главе 25.

Окружения Python

Типичный программист на Python работает одновременно над несколькими проектами, каждый из которых имеет собственный список зависимостей (как правило, это сторонние библиотеки и файлы данных). Если зависимости для всех проектов установлены в одном и том же интерпретаторе Python, то очень трудно определить, какие зависимости используются теми или иными проектами, и вовсе невозможно управлять проектами, где некоторые зависимости конфликтуют из-за различий в версиях.

Ранние интерпретаторы Python создавались в предположении, что каждая компьютерная система будет иметь установленный в ней "интерпретатор Python", который будет использоваться для обработки любого кода Python, выполняющегося в этой системе. Дистрибутивы операционных систем стали включать Python в свою базовую установку, но поскольку Python активно развивался, от пользователей начали поступать жалобы на то, что они хотели бы использовать более современную версию языка, чем та, которая предоставляется операционной системой.

Возникли методики, позволяющие устанавливать в системе несколько версий языка, но сам процесс установки оставался нестандартным и требовал вмешательства в систему. Эта проблема была частично разрешена введением каталога *site-packages* в качестве репозитория модулей, добавляемых в установку Python, но поддержка использования проектов с конфликтующими требованиями в одном и том же интерпретаторе по-прежнему оставалась невозможной.

Программистам, привыкшим работать с командной строкой, знакомо понятие *окружение оболочки*. Программа оболочки, выполняющаяся в процессе, имеет текущий каталог, переменные, которые могут устанавливаться командами оболочки (что похоже на пространство имен Python), и другие компоненты специфических для процесса данных. Программы на Python получают доступ к окружению оболочки посредством модуля `os.environ`.

Как отмечалось в разделе "Переменные среды" главы 2, на работу Python оказывают влияние различные аспекты окружения оболочки. Например, то, какой интерпретатор будет выполнен в ответ на команду `python` и другие команды, определяется переменной среды `PATH`. Вы вправе рассматривать эти аспекты окружения оболочки, влияющие на выполнение операций в Python, как ваше *окружение Python*. Изменяя его, вы сможете устанавливать, какой интерпретатор Python будет запускаться в ответ на команду `python`, какие пакеты и модули доступны под определенными именами и пр.

Оставьте установленный системой Python для самой системы

Мы рекомендуем вам управлять своим окружением Python. В частности, не создавайте приложения поверх Python, распространяемого вместе с системой. Вместо этого установите другой независимый дистрибутив Python и настройте свое окружение оболочки, чтобы команда `python` запускала вашу локальную установку Python, а не системную.

Используйте виртуальное окружение

Утилита `pip` обеспечила простой способ установки (и впервые — отмены установки) пакетов и модулей в окружении Python. Изменение каталога *site-packages* системного варианта Python по-прежнему требует использования административных привилегий, и того же самого требует выполнение утилиты `pip` (хотя при желании вы сможете выполнить установку в каталог, отличный от *site-packages*), а установленные модули видны всем программам.

Недостающим элементом является возможность внесения контролируемых изменений в окружение Python, обеспечивающих использование конкретного интерпретатора и конкретного набора библиотек Python. Именно эту возможность предоставляют вам *виртуальные окружения*. Виртуальное окружение, создаваемое на основе конкретного интерпретатора Python, копирует компоненты из установки данного интерпретатора или создает ссылки на них. Критично, однако, то, что каждое окружение должно иметь собственный каталог *site-packages*, в который вы сможете устанавливать выбранные вами ресурсы.

Процедура создания виртуального окружения намного проще процедуры установки Python и требует гораздо меньше системных ресурсов (типичное вновь создаваемое виртуальное окружение занимает менее 20 Мбайт). Вы сможете легко создавать и активизировать их по мере необходимости и столь же легко деактивизировать и уничтожать. Виртуальное окружение можно активизировать и деактивизировать сколько угодно раз на протяжении времени его жизни, а если необходимо обновить установленные ресурсы, то можно использовать утилиту `pip`. Удаление дерева каталогов виртуального окружения, когда в нем больше нет необходимости, полностью освобождает занимаемое им дисковое пространство. Время жизни виртуального окружения может исчисляться как минутами, так и месяцами.

Что такое виртуальное окружение

По сути, виртуальное окружение — это автономное подмножество вашего окружения Python, которое можно активизировать и деактивизировать по мере необходимости. Для интерпретатора Python X.Y оно включает, среди всего прочего, каталог *bin*, содержащий интерпретатор Python X.Y, и каталог *lib/pythonX.Y/site-packages*, содержащий предустановленные версии `easyinstall`, `pip`, `pkg_resources` и `setuptools`. Сопровождение отдельных копий этих важных ресурсов, имеющих отношение к распространению пакетов, позволяет обновлять их по мере необходимости, а не вынужденно привязываться к базовому дистрибутиву Python.

Виртуальное окружение располагает собственными копиями (на платформе Windows) дистрибутивных файлов Python или символическими ссылками на них (на других платформах). Оно настраивает значения `sys.prefix` и `sys.exec_prefix`, по которым интерпретатор и различные установочные утилиты определяют местоположение некоторых библиотек. Это означает, что утилита `pip` может устанавливать

зависимости в каталогах *site-packages* виртуального окружения, изолируя их от других окружений. В результате виртуальное окружение переопределяет, какой интерпретатор должен запускаться, когда вы выполняете команду `python`, и какие библиотеки ему доступны, но оставляет нетронутыми большинство других аспектов вашего окружения Python (таких, как переменные `PYTHONPATH` и `PYTHONHOME`). Поскольку его изменение влияет на ваше окружение оболочки, оно также оказывает влияние на любые подоболочки, в которых вы выполняете команды.

Имея разные окружения, вы сможете, например, тестировать в проекте две различные версии одной и той же библиотеки или тестировать свой проект в различных версиях Python (весьма полезно для проверки совместимости вашего кода с версиями v2/v3). Кроме того, вы можете добавлять зависимости в свои проекты Python, даже не обладая особыми привилегиями, поскольку обычно будете создавать свои виртуальные окружения в тех местах, в которые вам разрешена запись.

В течение длительного времени единственным способом создания виртуальных окружений было использование стороннего пакета `virtualenv` (`https://virtualenv.pypa.io/en/stable/`) с поддержкой или без поддержки со стороны пакета `virtualenvwrapper` (`https://virtualenvwrapper.readthedocs.io/en/latest/`), причем оба пакета все еще доступны для версии v2. Подробнее об этих инструментах можно прочитать в руководстве пользователя по созданию пакетов (`packaging.python.org`). Эти инструменты работают также в версии v3, но в выпуске 3.3 добавлен модуль `venv`, впервые делающий виртуальные окружения собственным средством Python.

Новое в Python 3.6: используйте команду **`python -m venv`** ***`envpath`*** вместо команды **`pyvenv`**, которая в настоящее время признана устаревшей.

Создание и удаление виртуальных окружений

В версии v3 команда **`python -m venv`** ***`envpath`*** создает виртуальное окружение (в каталоге *envpath*, который создается в случае необходимости) на основе интерпретатора Python, использованного при выполнении команды. Указав несколько каталогов в виде аргументов, можно создать с помощью одной команды несколько виртуальных окружений, в которые вы затем установите различные наборы зависимостей. Команда `venv` принимает ряд опций, перечисленных в табл. 6.1.

Таблица 6.1. Опции команды `venv`

Опция	Назначение
`--clear`	Удаляет содержимое каталога, прежде чем установить виртуальное окружение
`--copies`	Устанавливает файлы посредством копирования на Unix-подобных платформах, на которых по умолчанию используются символические ссылки

Окончание табл. 6.1

Опция	Назначение
`-h` или `--help`	Выводит краткую информацию о синтаксисе использования команды и список доступных опций
`--system-site-packages`	Добавляет в список путей поиска виртуального окружения стандартный системный каталог *site-packages*, тем самым делая модули базовой установки Python доступными в виртуальном окружении
`--symlinks`	Устанавливает файлы путем использования символических ссылок на платформах, на которых по умолчанию используется копирование файлов
`--upgrade`	Устанавливает выполняющуюся версию Python в виртуальном окружении, заменяя версию окружения, с которой оно создавалось
`--without-pip`	Отменяет обычное поведение, заключающееся в вызове команды `ensurepip` для загрузки установочной утилиты `pip` в виртуальное окружение

Пользователи версии v2 должны использовать команду **`python -m virtualenv`**, которая не допускает указания нескольких каталогов в качестве аргументов.

В качестве примера ниже показан сеанс работы за терминалом, демонстрирующий процесс создания виртуального окружения и структуру созданного дерева каталогов. Как следует из листинга подкаталогов каталога *bin*, данный конкретный пользователь по умолчанию использует интерпретатор версии v3, установленный в каталоге */usr/local/bin*.

```
machine:~ user$ python3 -m venv /tmp/tempenv
machine:~ user$ tree -dL 4 /tmp/tempenv
/tmp/tempenv
├── bin
├── include
└── lib
    └── python3.5
        └── site-packages
            ├── __pycache__
            ├── pip
            ├── pip-8.1.1.dist-info
            ├── pkg_resources
            ├── setuptools
            └── setuptools-20.10.1.dist-info

11 directories
machine:~ user$ ls -l /tmp/tempenv/bin/
total 80
-rw-r--r-- 1 sh wheel 2134 Oct 24 15:26 activate
```

```
-rw-r--r-- 1 sh wheel 1250 Oct 24 15:26 activate.csh
-rw-r--r-- 1 sh wheel 2388 Oct 24 15:26 activate.fish
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install-3.5
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3.5
lrwxr-xr-x 1 sh wheel    7 Oct 24 15:26 python->python3
lrwxr-xr-x 1 sh wheel   22 Oct 24 15:26 python3->/usr/local/bin/python3
```

Удаление виртуального окружения сводится к удалению каталога, в котором оно находится (а также всех подкаталогов и файлов в дереве: **`rm -rf envpath`** в случае Unix-подобных систем). Простота удаления — полезное свойство виртуальных окружений.

Модуль `venv` включает средства, облегчающие программное создание настраиваемых виртуальных окружений (например, путем предварительной установки определенных модулей в окружение или выполнения других необходимых операций после того, как окружение было установлено). Все эти возможности самым исчерпывающим образом описаны в онлайн-документации (`https://docs.python.org/3/library/venv.html`), а потому рассмотрение данного API в нашей книге ограничивается вышеприведенным обсуждением.

Работа с виртуальными окружениями

Чтобы использовать виртуальное окружение, вы должны *активизировать* его из обычного окружения своей оболочки. В любой момент времени может существовать только одно виртуальное окружение — операции активизации не укладываются в "стек", подобно вызовам функций. Активизация подготавливает ваше окружение Python к использованию интерпретатора Python виртуального окружения и каталога *site-packages* (вместе с полной стандартной библиотекой). Если вы хотите прекратить использование этих зависимостей, деактивизируйте виртуальное окружение, что сделает вновь доступным ваше стандартное окружение Python. Дерево каталогов виртуального окружения продолжает существовать до тех пор, пока вы не удалите его, поэтому вы вольны активизировать и деактивизировать виртуальное окружение по своему усмотрению.

Активизация виртуального окружения на Unix-подобных платформах требует использования команды оболочки `source`, чтобы команды скрипта активизации могли внести изменения в текущее окружение оболочки. Простой запуск скрипта означал бы, что его команды будут выполнены в подоболочке, а после выхода из подоболочки все изменения будут утеряны. В случае `bash` и других аналогичных оболочек для активизации окружения в каталоге *envpath* следует выполнить такую команду:

```
source envpath/bin/activate
```

Пользователи других оболочек могут воспользоваться скриптами *activate.csh* и *activate.fish*, находящимися в том же каталоге. В случае Windows-систем используйте такую команду:

***envpath*\Scripts\activate.bat**

В процессе активизации выполняется ряд операций, наиболее важными из которых являются следующие:

- добавление каталога `bin` виртуального окружения в начало переменной среды `PATH` оболочки, чтобы его команды имели приоритет относительно других команд с тем же именем, уже имеющимся в списке `PATH`;
- определение команды деактивизации для удаления всех последствий активизации и возврата окружения Python в первоначальное состояние;
- изменение приглашения оболочки и включение в него имени виртуального окружения;
- определение переменной среды `VIRTUAL_ENV` в качестве пути доступа к корневому каталогу виртуального окружения (сценарии могут использовать эту информацию для интроспекции виртуального окружения).

В результате выполнения этих действий, как только активизируется виртуальное окружение, команда `python` будет запускать ассоциированный с ним интерпретатор. Интерпретатор видит библиотеки (модули и пакеты), которые были установлены в этом виртуальном окружении, а утилита `pip`, которая теперь берется из виртуального окружения, поскольку в процессе установки модуля она также устанавливается в каталоге `bin` виртуального окружения, по умолчанию устанавливает новые пакеты и модули в каталоге *site-packages* виртуального окружения.

Те, кто впервые сталкивается с виртуальным окружением, должны понимать, что оно не связано с каталогом какого-либо проекта. Вполне допускается работать сразу с несколькими проектами, каждый из которых имеет собственное дерево исходных файлов, используя одно и то же виртуальное окружение. Активизируйте его, а затем свободно перемещайтесь между различными местами хранения файлов, и при этом вам будут доступны одни и те же библиотеки (поскольку окружение Python определяется виртуальным окружением).

Когда вы захотите отключить виртуальное окружение и перестать использовать данный набор ресурсов, выполните команду `deactivate`.

Этого будет достаточно для того, чтобы отменить изменения, внесенные командой активизации, путем удаления каталога `bin` виртуального окружения из переменной `PATH`, после чего команда `python` будет вновь запускать ваш обычный интерпретатор. Коль скоро вы не удаляете виртуальное окружение, оно остается доступным для будущего использования путем повторения действий, необходимых для его активизации.

Управление требованиями зависимостей

Ввиду того что виртуальные окружения изначально предназначались для установки с помощью `pip`, вы не будете удивлены тем, что использование утилиты `pip` является предпочтительным способом обслуживания зависимостей в виртуальном окружении. Поскольку утилита `pip` уже имеет обширную документацию, мы ограничиваемся лишь упоминанием о том, чего достаточно для того, чтобы продемонстрировать преимущества ее использования в виртуальных окружениях. Создав виртуальное окружение, активизировав его и установив зависимости, вы сможете определить точные версии этих зависимостей с помощью команды `pip freeze`.

```
(tempenv) machine:~ user$ pip freeze
appnope==0.1.0
decorator==4.0.10
ipython==5.1.0
ipython-genutils==0.1.0
pexpect==4.2.1
pickleshare==0.7.4
prompt-toolkit==1.0.8
ptyprocess==0.5.1
Pygments==2.1.3
requests==2.11.1
simplegeneric==0.8.1
six==1.10.0
traitlets==4.3.1
wcwidth==0.1.7
```

Перенаправив вывод этой команды в файл *имя_файла*, вы сможете воссоздать тот же набор зависимостей в другом виртуальном окружении с помощью команды **`pip install -r` *`имя_файла`***.

Распространяя код, предназначенный для использования другими людьми, разработчики на Python обычно включают в него файл *requirements.txt*, в котором перечисляют необходимые зависимости. Когда вы устанавливаете программное обеспечение из репозитория Python Package Index, `pip` устанавливает затребованные вами пакеты вместе со всеми указанными зависимостями. В процессе разработки программ удобно иметь файл требований, который можно использовать для добавления необходимых зависимостей в активное виртуальное окружение (если только они уже не установлены) с помощью простой команды **`pip install -r requirements.txt`**.

Для поддержания одного и того же набора зависимостей в нескольких виртуальных окружениях добавляйте зависимости в каждое из них, используя один и тот же файл требований. Это общепринятый способ разработки проектов, предназначенных для выполнения в нескольких версиях Python: создайте виртуальное окружение на основе одной из требуемых версий, а затем установите в каждое из них зависимости из одного и того же файла требований. В то время как в предыдущем примере использовались точно версионированные спецификации зависимостей, полученные

с помощью команды `pip freeze`, на практике для определения требований к зависимостям и ограничениям можно использовать довольно сложные способы.

Лучшие практики использования виртуальных окружений

Несмотря на то что в Интернете без труда можно найти массу полезных рекомендаций по этому поводу, можно дать один небольшой, но тем не менее замечательный совет относительно того, как лучше всего организовать работу с виртуальными окружениями.

Работая с одними и теми же зависимостями в нескольких версиях Python, полезно указывать номер версии в имени окружения и использовать общий префикс. Таким образом, для проекта *mutex* можно было бы поддерживать окружения *mutex_35* и *mutex_27* для разработки в версиях v3 и v2 соответственно. Когда вам совершенно ясно, для какой версии Python предназначен проект (вспомните, что имя окружения отображается в приглашении оболочки), вероятность тестирования приложения в неподходящей версии значительно снижается. Для управления установкой ресурсов в обоих окружениях используйте общий файл требований.

Регистрируйте только файл (файлы) требований в системе управления версиями (Source Code Control System, SCCS), а не все окружение. Имея файл требований, можно легко воссоздать виртуальное окружение, зависящее только от выпуска Python и требований. Вы распространяете свой проект и предоставляете пользователям самостоятельно решать, в какой версии (версиях) Python его выполнять, создавая подходящее (предпочтительно виртуальное) окружение.

Храните свои виртуальные окружения вне каталогов проектов. Это позволит избежать необходимости явным образом вынуждать систему управления версиями игнорировать их. В действительности не имеет значения, где хранятся окружения, — система `virtualenvwrapper` сохранит их в централизованном расположении.

Ваше окружение Python не зависит от местонахождения вашего проекта в файловой системе. Вы можете активизировать виртуальное окружение, а затем переключаться между ветвями и перемещаться по дереву изменений в SCCS, чтобы использовать окружение там, где вам нужно.

Чтобы исследовать новый модуль или пакет, создайте и активизируйте новое виртуальное окружение, а затем установите интересующие вас ресурсы с помощью команды `pip install`. Вы вольны экспериментировать с этим новым окружением, как вам заблагорассудится, будучи уверенным в том, что не установите неподходящие зависимости в другие проекты.

Может оказаться так, что для экспериментов в виртуальном окружении приходится устанавливать ресурсы, которые не требуются в текущем проекте. Вместо того чтобы засорять свою среду разработки, разветвите ее: создайте новое виртуальное окружение на основе тех же требований плюс тестируемая функциональность.

Впоследствии, для того чтобы сделать эти изменения постоянными, используйте SCCS для обратного слияния изменений исходного кода и требований из ветви.

По желанию можно создавать виртуальные окружения на основе отладочных сборок Python, что предоставит вам доступ к обильной инструментальной информации о производительности вашего кода на Python (и, разумеется, интерпретатора).

Сама разработка собственных виртуальных окружений требует контроля изменений, и простота их создания способствует этому. Предположим, вы недавно выпустили версию 4.3 модуля и хотите протестировать его код с новыми версиями двух его зависимостей. Обладая достаточными навыками, вы *могли бы* заставить `pip` заменить имеющиеся зависимости в вашем существующем виртуальном окружении.

Однако гораздо легче разветвить проект, обновить требования и создать совершенно новое виртуальное окружение на основе обновленных требований. Вы по-прежнему имеете нетронутое исходное виртуальное окружение и можете переключаться между окружениями для исследования специфических аспектов любых аспектов миграции, которые могут возникать. Как только вы адаптируете код таким образом, чтобы все тесты проходили с обновленными зависимостями, вы фиксируете изменения своего кода и требований и выполняете слияние в версию 4.4, чтобы завершить обновление, известив своих коллег о том, что теперь ваш код готов для работы с обновленными версиями зависимостей.

Виртуальные окружения не решают всех проблем, возникающих перед программистом на языке Python. Всегда можно создать более сложные или более общие инструменты. Но, слава Богу, виртуальные окружения работают, и мы должны как можно полнее использовать все предлагаемые ими преимущества.