7 Совместная работа

В Python предусмотрены средства, облегчающие создание четко определенных интерфейсов прикладного программирования (API) с отчетливыми границами между ними. Сообщество Python выработало принципы наилучшей практики, которые максимально облегчают обслуживание кода в процессе его эксплуатации. Также существуют стандартные инструменты, поставляемые с Python, которые обеспечивают совместную работу команд программистов, работающих в разных средах.

Совместная работа над программами Python требует тщательного подхода к стилю написания кода. Даже если вы работаете независимо, существует большая вероятность того, что вы будете использовать код, содержащийся в стандартных библиотеках или пакетах с открытым исходным кодом, написанных другими людьми. Очень важно, чтобы вам были понятны механизмы, упрощающие сотрудничество с другими программистами на языке Python.

Рекомендация 49. Снабжайте строками документирования каждую функцию, класс и модуль

Документация играет чрезвычайно важную роль в Python ввиду динамичной природы этого языка. Python предоставляет встроенную поддержку присоединения документации к блокам кода. В отличие от многих других языков, документация из исходного кода программы доступна непосредственно во время выполнения программы.

Например, вы можете добавить так называемые *строки документирования* (docstrings) сразу же за инструкцией `def` функции.

```
def palindrome(word):
    """Возвращает True, если заданное слово является
       палиндромом."""
    return word == word[::-1]
```

Строки документирования можно извлекать из самой программы на Python с помощью специального атрибута функции, который называется `__doc__`.

```
print(repr(palindrome.__doc__))

>>>
Возвращает True, если заданное слово является палиндромом.'
```

Строки документирования могут присоединяться к функциям, классам и модулям, становясь частью процесса компиляции и выполнения программы. Поддержка этого средства и атрибута `__doc__` имеет три следствия.

- Легкодоступная документация упрощает интерактивную разработку. Используя встроенную функцию `help()`, можно проверять наличие строк документирования в функциях, классах и модулях. Благодаря этому разработка алгоритмов, тестирование прикладных программных интерфейсов и написание небольших фрагментов кода с помощью интерактивного интерпретатора Python ("оболочки" Python) и таких средств, как IPython Notebook (`http://ipython.org`), доставляет одно удовольствие.
- Наличие стандартного способа подготовки документации упрощает создание инструментов, преобразующих текст в более приемлемые форматы (например, HTML). Благодаря этому сообщество пользователей Python получило в свое распоряжение такие великолепные средства генерирования документации, как Sphinx (`http://sphinxdoc.org`). Кроме того, это подтолкнуло к созданию основанных сообществом сайтов, таких как Read the Docs (`https://readthedocs.org`), которые бесплатно предоставляют возможность размещения хорошо подготовленной документации для проектов Python с открытым исходным кодом.
- Первоклассная, легкодоступная и отлично выглядящая документация Python поощряет людей к написанию еще большего количества документации такого же качества. Члены сообщества Python убеждены в важности документирования кода. Они считают, что понятие "хороший код" означает, кроме всего прочего, хорошо документированный код. Это означает, что в большинстве случаев вы можете рассчитывать на то, что библиотеки Python снабжены отличной документацией.

Чтобы приобщиться к этой великолепной культуре документирования кода, вы должны следовать нескольким рекомендациям, касающимся написания строк Dostrings. Более подробное обсуждение этого

вопроса вы найдете на сайте PEP 257 (http://www.python.org/dev/peps/pep-0257/). Также существуют стандарты оформления строк документирования, которые вы должны обязательно соблюдать.

Документирование модулей

Каждый модуль должен снабжаться строками документирования верхнего уровня. Этот строковый литерал должен быть первой инструкцией в файле исходного кода, а также начинаться и заканчиваться тремя двойными кавычками ("""). В него помещают сведения о модуле и его содержимом ознакомительного характера.

Первая строка документирования должна содержать единственное предложение, описывающее назначение модуля. В следующих абзацах должно содержаться подробное описание работы модуля, рассчитанное на пользователей. Строки документирования модуля служат отправным пунктом, в котором можно отдельно выделить важные классы и функции, принадлежащие модулю.

Ниже приведен пример строки документирования модуля.

```
# words.py
#!/usr/bin/env python3
"""Библиотека для тестирования слов с помощью различных
   лингвистических шаблонов.

Тестирование отношений между словами иногда может быть затруднено!
Данный модуль предоставляет методы, позволяющие легко определить,
какие из найденных слов имеют специальные свойства.

Доступные функции:
- palindrome: определяет, является ли слово палиндромом.
- check_anagram: определяет, являются ли данная пара слов
  анаграммой.
...
"""

# ...
```

Если модуль является утилитой командной строки, то строки документирования — это отличное место для размещения информации о порядке запуска данной утилиты.

Документирование классов

Каждый класс должен снабжаться строками документирования уровня класса. В основном эти строки следуют тому же шаблону, что и строки документирования уровня модуля. Первая строка должна состоять

из одного предложения и содержать описание назначения класса. В последующих абзацах приводятся наиболее важные детали того, как работает этот класс.

В строках документирования уровня класса должны быть отдельно указаны открытые атрибуты и методы. Они также должны содержать рекомендации относительно того, как подклассы должны взаимодействовать с защищенными атрибутами (см. раздел "Рекомендация 27. Предпочитайте открытые атрибуты закрытым") и методами суперкласса.

Ниже приведен пример строки документирования класса.

```
class Player(object):
    """Представляет игрока в игре.

    Подклассы могут переопределять метод 'tick', обеспечивающий
    пользовательскую анимацию движений игрока в зависимости от
    запаса силы и т.п.
Открытые атрибуты:
- power: неизрасходованная сила(значение с плавающей запятой
         (в интервале от 0 до 1).
- coins: количество найденных монет на данном уровне
         (целочисленное значение).
"""
# ...
```

Документирование функций

Каждая общедоступная функция и метод должны снабжаться строками документирования. Эти строки должны следовать тому же шаблону, что и модули и классы. Первая строка должна состоять из одного предложения и содержать описание функции. Последующие абзацы должны содержать описание специфического поведения и аргументов функции. Также должны быть указаны возвращаемые значения и описаны исключения, которые вызывающий код должен обрабатывать как часть интерфейса функции.

Ниже приведен пример строки документирования функции.

```
def find_anagrams(word, dictionary):
"""Находит все анаграммы для слова.

Эта функция выполняется лишь со скоростью выполнения теста,
проверяющего членство в контейнере 'dictionary'.
Она будет работать медленно, если 'dictionary' - список,
и быстро, если 'dictionary' - множество.
```

```
Аргументы:
word: строка целевого слова.
dictionary: контейнер, содержащий все строки, о которых
известно, что они представляют собой действительные слова.

Возвращаемые значения:
Список найденных анаграмм. Если ни одно слово не найдено,
этот список пустой.
"""
# ...
```

Кроме того, при написании строк документирования для функций встречаются особые случаи, о которых необходимо знать.

- Если ваша функция не имеет аргументов, а возвращаемое значение — простое, то вполне допустимо описать ее одним предложением.
- Если ваша функция ничего не возвращает, то лучше вообще не упоминать о возвращаемом значении, чем писать "возвращает `None`".
- Если вы не ожидаете, что ваша функция может сгенерировать исключение в процессе нормальной работы, то не пишите об этом.
- Если ваша функция принимает переменное количество аргументов (см. раздел "Рекомендация 18. Снижайте визуальный шум с помощью переменного количества позиционных аргументов") или именованные аргументы (см. "Рекомендация 19. Обеспечивайте опциональное поведение с помощью именованных аргументов"), то используйте выражения `*args` и `**kwargs` в документированном списке аргументов для описания их назначения.
- Если ваша функция имеет аргументы с заданными по умолчанию значениями, то эти значения по умолчанию должны быть указаны (см. раздел "Рекомендация 20. Используйте значение `None` и средство `Docstrings` при задании динамических значений по умолчанию для аргументов").
- Если ваша функция — генератор (см. раздел "Рекомендация 16. Не упускайте возможность использовать генераторы вместо возврата списков"), то строки документирования должны описывать, что именно возвращает генератор в процессе итерирования.
- Если ваша функция — сопрограмма (см. раздел "Рекомендация 40. Используйте сопрограммы для одновременного выполнения нескольких функций"), то строки документирования должны содержать описание того, что возвращает сопрограмма, что она ожида-

ет получить из выражения `yield` и когда останавливается итерационный процесс.

Примечание

После того как вы запишете строки документирования для своих модулей, важно следить за обновлением документации. Встроенный модуль `doctest` упрощает выполнение контрольных примеров, включаемых в строки документирования для того, чтобы можно было убедиться в том, что между исходным кодом и документацией со временем не возникают расхождения.

Что следует запомнить

- Пишите документацию для каждого модуля, класса и функции, используя строки документирования кода. Обновляйте документацию по мере внесения изменений в код.
- Для модулей: кратко описывайте содержимое модуля, выделяя важные классы функции, о которых должны знать пользователи.
- Для классов: документируйте поведение, важные атрибуты и поведение подклассов в строках документирования, помещая их вслед за инструкциями `class`.
- Для функций и методов: документируйте каждый аргумент, возвращаемое значение, генерируемое исключение и другие аспекты поведения в строках документирования, помещая их вслед за инструкцией `def`.

Рекомендация 50. Используйте пакеты для организации модулей и предоставления стабильных API

Вполне естественно, что по мере роста кодовой базы программ вы сталкиваетесь с необходимостью реорганизации ее структуры. С этой целью вы разбиваете крупные функции на более мелкие, преобразуете структуры данных во вспомогательные классы (см. раздел “Рекомендация 22. Отдавайте предпочтение структуризации данных с помощью классов, а не словарей или кортежей”) и разделяете функциональность между различными модулями, которые зависят друг от друга.

В какой-то момент вы обнаружите, что число модулей настолько возросло, что для сохранения ясности структуры кода требуется вводить дополнительный программный слой. В Python эта задача решается с помощью *пакетов* — модулей, содержащих другие модули.

В большинстве случаев пакеты определяют, помещая в каталог пустой файл с именем *__init__.py*. После этого любой файл Python, находящийся в том же каталоге, что и файл *__init__.py*, становится доступным для импорта с использованием пути относительно данного каталога. Предположим, что ваша программа имеет показанную ниже структуру каталогов.

```
main.py
mypackage/__init__.py
mypackage/models.py
mypackage/utils.py
```

Чтобы импортировать модуль `utils`, вы используете абсолютное имя модуля, которое включает имя каталога пакета.

```
# main.py
from mypackage import utils
```

Эта же схема остается в силе и в тех случаях, когда каталоги пакетов располагаются в других пакетах (например, `mypackage.foo.bar`).

Примечание

В Python 3.4 введен более гибкий способ определения пакетов: пространства имен. Пространства имен могут включать модули из совершенно разных каталогов, zip-архивов и даже удаленных систем. За более подробными сведениями о пространствах имен обратитесь к документу PEP 420 (`http://www.python.org/dev/peps/pep-0420/`).

Предоставляемая пакетами функциональность преследует в программах на языке Python две основные цели, которые описаны ниже.

Пространства имен

Прежде всего, пакеты предназначены для того, чтобы облегчить группирование модулей по различным пространствам имен. Это позволяет иметь много модулей с одним и тем же именем, но различными путями доступа, которые являются уникальными. Например, ниже приведен фрагмент программы, которая импортирует атрибуты из двух модулей с одним и тем же именем `utils.py`. Это возможно, поскольку к модулям можно обращаться, используя их абсолютные пути.

```
# main.py
from analysis.utils import log_base2_bucket
from frontend.utils import stringify

bucket = stringify(log_base2_bucket(33))
```

Однако такой подход перестает работать, если функции, классы или подмодули определены в одноименных пакетах. Допустим, вы хотите использовать функцию `inspect()`, которая имеется как в модуле `analysis.utils`, так и в модуле `frontend.utils`. Непосредственное импортирование этих атрибутов не сработает, поскольку вторая инструкция `import` перекроет значение `inspect` в текущей области видимости.

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect # Перекрывает!
```

Решение заключается в использовании предложения `as` инструкции `import` для переименования объектов, импортируемых в текущую область видимости.

```
# main3.py
from analysis.utils import inspect as analysis_inspect
from frontend.utils import inspect as frontend_inspect

value = 33
if analysis_inspect(value) == frontend_inspect(value):
    print('Равенство!')
```

Предложение `as` можно использовать для переименования всего, что вы получаете с помощью инструкции `import`, включая целые модули. Это упрощает доступ к коду в пространствах имен и делает его идентификацию понятной в процессе его использования.

Примечание

Другой подход, позволяющий избежать конфликта между импортированными именами, заключается в том, чтобы при обращении к именам всегда использовать уникальное имя модуля с наивысшим положением в иерархии.

Применительно к приведенному выше примеру вы прежде всего должны импортировать модули `analysis.utils` и `frontend.utils` с помощью инструкций `import`. После этого вы должны обращаться к функциям `inspect()`, используя полные пути доступа к ним: `analysis.utils.inspect()` и `frontend.utils.inspect()`.

Такой подход избавляет от необходимости использовать предложение `as`. Он также делает для новых читателей совершенно очевидным то, где именно определена каждая функция.

Стабильные API

Второе назначение пакетов в Python — это предоставление строго определенных, стабильных интерфейсов прикладного программирования (API) внешним потребителям.

Когда вы пишете API для широкого круга потребителей, например в виде пакета с открытым исходным кодом (см. раздел "Рекомендация 48: Знайте, где искать модули, разработанные сообществом Python"), вы хотите предоставить стабильную функциональность, которая не изменяется от выпуска к выпуску. Для этого очень важно скрывать организацию внутреннего кода от внешних пользователей. Это позволяет выполнять рефакторинг и улучшать внутренние модули пакета, не создавая при этом препятствий для их эксплуатации существующими пользователями.

Python может ограничивать открытость API для потребителей с помощью специального атрибута `__all__` модуля или пакета. Значением `__all__` является список всех имен, экспортируемых из модуля как часть его общедоступного API. Когда в коде, использующем модуль, выполняется инструкция `from foo import *`, из модуля `foo` импортируются лишь атрибуты, фигурирующие в списке `foo.__all__`. В случае отсутствия свойства `__all__` у модуля `foo` импортируются лишь общедоступные атрибуты — те, имя которых не начинается с символа подчеркивания (см. раздел "Рекомендация 27. Предпочитайте общедоступные атрибуты закрытым").

Допустим, вы хотите предоставить пакет для расчета столкновений между движущимися частицами. Определим модуль `models` из пакета `mypackage`, который будет содержать представление частиц.

```
# models.py
__all__ = ['Projectile']

class Projectile(object):
    def __init__(self, mass, velocity):
        self.mass = mass
        self.velocity = velocity
```

Мы также определим модуль `utils` в пакете `mypackage`, который будет выполнять различные операции над экземплярами `Projectile`, такие как имитация столкновений между ними.

```
# utils.py
from . models import Projectile

__all__ = ['simulate_collision']

def _dot_product(a, b):
    # ...

def simulate_collision(a, b):
    # ...
```

Теперь мне хотелось бы предоставить все общедоступные части этого API в виде набора атрибутов, доступных в модуле `mypackage`. Это позволит потребителям модуля всегда выполнять импорт непосредственно из пакета `mypackage` вместо того, чтобы импортировать из пакетов `mypackage.models` или `mypackage.utils`. Кроме того, это будет гарантировать нормальную работу кода, использующего API, даже если внутренняя организация пакета `mypackage` изменится (например, будет удален файл *models.py*).

Чтобы реализовать такой подход, потребуется изменить файл *__init__.py* в каталоге `mypackage`. Когда этот файл импортируется, он фактически становится содержимым модуля `mypackage`. Следовательно, вы можете указать явный API для пакета `mypackage`, налагая ограничения на то, что допускается импортировать в файл *__init__.py*. Поскольку все наши внутренние модули уже указывают `__all__`, можно представить общедоступный интерфейс `mypackage`, осуществляя весь импорт из внутренних модулей и соответственно обновляя список `__all__`.

```
# __init__.py
__all__ = []
from . models import *
__all__ += models.__all__
from . utils import *
__all__ += utils.__all__
```

Ниже приведен код, использующий этот API, который выполняет импорт непосредственно из пакета `mypackage`, а не обращается к внутренним модулям.

```
# api_consumer.py
from mypackage import *

a = Projectile(1.5, 3)
b = Projectile(4, 1.7)
after_a, after_b = simulate_collision(a, b)
```

Самое главное то, что функции, предназначенные исключительно для внутреннего использования, такие как `mypackage.utils._dot_product()`, не будут доступны потребителям API через пакет `mypackage`, поскольку они не были включены в список `__all__`. В силу этого они не импортируются инструкцией `from mypackage import *`. Имена, предназначенные для внутреннего использования, эффективно скрываются.

В целом такой подход отлично работает в ситуациях, когда важно предоставить явный, стабильный API. Но если вы создаете API, предназначенный для использования с собственными модулями, то в функциональности, предоставляемой атрибутом `__all__`, вероятно, нет необхо-

димости, и ее следует избегать. Обычно механизма пространства имен, обеспечиваемого пакетами, вполне достаточно для того, чтобы команда программистов могла коллективно работать с большими объемами кода, который они контролируют, при сохранении разумных границ интерфейсов.

Остерегайтесь использовать инструкцию `import *`

Инструкции импорта типа `from x import y` совершенно понятны, поскольку в качестве источника `y` в них явно указывается пакет или модуль `x`. Импорт с использованием групповых символов, как в инструкции `from foo import *`, также может быть полезен, особенно в интерактивных сеансах работы с Python. Однако использование групповых символов затрудняет понимание кода.

- Инструкция `from foo import *` скрывает источник имен от тех, кто впервые читает код. Если в модуле имеется несколько инструкций `import *`, то для того, чтобы разобраться, где определено то или иное имя, приходится просматривать все модули, на которые имеются ссылки.
- Имена из инструкций `from import *` будут перекрывать любые конфликтующие имена в пределах вмещающего модуля. Это может приводить к странным ошибкам, обусловленным случайным взаимодействием между вашим кодом и перекрывающимися именами из нескольких инструкций `import *`.

Наиболее безопасный подход заключается в том, чтобы избегать использования инструкций `import *` в своем коде и явно импортировать имена с помощью инструкций типа `from x import y`.

Что следует запомнить

- ✦ В языке Python пакеты — это модули, которые содержат другие модули. Пакеты позволяют организовать код в виде отдельных пространств имен с уникальными абсолютными именами модулей, исключающими возможность возникновения конфликтов имен.
- ✦ Простые пакеты определяются путем добавления файла *__init__.py* в каталог, содержащий другие файлы с исходным кодом. Эти файлы становятся дочерними модулями пакета данного каталога. Каталоги пакетов могут содержать другие пакеты.
- ✦ Вы можете предоставить явный API для модуля, перечислив его общедоступные имена в специальном атрибуте `__all__`.

- ✦ Вы можете скрыть внутреннюю реализацию пакета, импортируя в файле *__init__.py* пакета лишь общедоступные имена или используя имена внутренних членов, начинающиеся с символа подчеркивания.
- ✦ В случае коллективной работы в рамках одной команды или над одной кодовой базой в использовании атрибута `__all__` для определения явных API, вероятно, нет необходимости.

Рекомендация 51. Изолируйте вызывающий код от API, определяя базовое исключение `Exception`

В процессе определения API модуля генерируемые вами исключения являются точно такой же частью вашего интерфейса, как и определяемые вами функции и классы (см. раздел "Рекомендация 14. Использование исключений предпочтительнее возврата значения `None`").

В Python имеется встроенная иерархия исключений для языка и стандартной библиотеки. Наблюдается тенденция использовать встроенные типы исключений для вывода сообщений об ошибках вместо определения собственных новых типов. Например, вы можете генерировать исключение `ValueError` всякий раз, когда функции передается некорректный параметр.

```
def determine_weight(volume, density):
    if density <= 0:
        raise ValueError('Плотность должна иметь положительное
↳ значение')
    # ...
```

В некоторых случаях в использовании исключения `ValueError` есть смысл, но для API намного эффективнее определять собственную иерархию исключений. Это можно сделать, предоставляя корневой объект `Exception` в своем модуле. Все остальные исключения, генерируемые в данном модуле, могут наследовать от данного корневого исключения.

```
# my_module.py
class Error(Exception):
    """Базовый класс для всех исключений, генерируемых
       данным модулем."""

class InvalidDensityError(Error):
    """Возникла проблема с предоставлением значения density."""
```

Определение корневого исключения в модуле упрощает коду, использующему ваш API, перехват исключений, которые генерируются вами

намеренно. Например, приведенный ниже код, использующий ваш API, вызывает функцию с помощью инструкции try/except, перехватывающей корневое исключение.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.Error as e:
    logging.error('Неожиданная ошибка: %s', e)
```

Данная инструкция try/except не дает исключениям, возникающим в вашем API, всплывать слишком высоко и тем самым нарушать работу вызывающей программы. Она изолирует вызывающий код от вашего API. Подобная изоляция обладает тремя преимуществами.

Во-первых, корневые исключения позволяют вызывающему коду распознавать ситуации, в которых проблема связана с использованием API. При соответствующем использовании вашего API вызывающий код может перехватывать и обрабатывать различные исключения, которые вы генерируете намеренно. Необработанные исключения распространяются вверх до изолирующего блока except, который перехватывает корневое исключение вашего модуля. Этот блок может известить потребителя API об исключении, предоставляя ему возможность добавить обработку исключений данного типа.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Ошибка в вызывающем коде: %s', e)
```

Вторым преимуществом использования корневых исключений является то, что это облегчает нахождение ошибок в коде модуля API. Если ваш код намеренно генерирует лишь те исключения, которые вы определили в иерархии модуля, то это означает, что все остальные типы исключений, сгенерированных в вашем модуле, относятся к категории тех, которые вы не намеревались генерировать, и именно они указывают на ошибки в вашем коде.

Использование инструкции try/except, как в приведенном выше примере, не будет изолировать потребителей API от ошибок в коде вашего API. Чтобы обеспечить такую изоляцию, вызывающий код должен добавить еще один блок except, который перехватывает исключения, соответствующие базовому классу Exception Python. Это позволит потребителям API обнаруживать в реализации модуля ошибки, подлежащие устранению.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Ошибка в вызывающем коде: %s', e)
except Exception as e:
    logging.error('Ошибка в коде API: %s', e)
    raise
```

Третье преимущество использования корневых исключений касается возможностей будущих проверок вашего API. Со временем у вас может возникнуть потребность в расширении API с целью предоставления более специфических исключений в определенных ситуациях. Например, вы можете добавить подкласс `Exception`, индицирующий состояние ошибки в случае предоставления отрицательных значений параметра `density`.

```
# my_module.py
class NegativeDensityError(InvalidDensityError):
    """Было предоставлено отрицательное значение
       параметра density."""

def determine_weight(volume, density):
    if density < 0:
        raise NegativeDensityError
```

Вызывающий код будет работать в точности так, как и прежде, поскольку он уже перехватывает исключения `InvalidDensityError` (родственный по отношению к исключению `NegativeDensityError` класс). В будущем вызывающий код может выделить новый тип исключения в качестве специального случая и соответственно изменить свое поведение.

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.NegativeDensityError as e:
    raise ValueError('Должно быть предоставлено неотрицательное
↳ значение плотности') from e
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error as e:
    logging.error('Ошибка в вызывающем коде: %s', e)
except Exception as e:
    logging.error('Ошибка в коде API: %s', e)
raise
```

Возможности организации проверок в коде в будущем можно расширить, предоставляя более широкий набор исключений непосредственно под корневым исключением. Допустим, у вас было предусмотрено несколько наборов ошибок, один из которых связан с расчетом веса (`weight`), другой — с расчетом объема (`volume`) и третий — с расчетом плотности (`density`).

```
# my_module.py
class WeightError(Error):
    """Базовый класс для ошибок при расчете веса."""

class VolumeError(Error):
    """Базовый класс для ошибок при расчете объема."""

class DensityError(Error):
    """Базовый класс для ошибок при расчете плотности."""
```

Специфические исключения будут наследовать от этих общих исключений. Каждое промежуточное исключение действует в качестве разновидности корневого исключения. Такой подход упрощает изолирование слоев вызывающего кода от кода API, основанного на широкой функциональности. Это гораздо лучше, чем заставлять вызывающий код каждый раз перехватывать длинный список весьма специфических исключений, соответствующих подклассам `Exception`.

Что следует запомнить

- Определение корневых исключений для модулей позволяет потребителям ваших API изолировать себя от них.
- Перехват корневых исключений может облегчить нахождение ошибок в коде, использующем ваш API.
- Перехват исключений базового класса `Exception` Python может облегчить нахождение ошибок в ваших реализациях API.
- Промежуточные корневые исключения позволяют добавлять специфические типы исключений в будущем, не нарушая работы кода, использующего ваш API.

Рекомендация 52. Знайте, как устранять циклические зависимости

В процессе коллективной разработки вы неизбежно столкнетесь с возникновением зависимостей между модулями. Это может происходить даже в тех случаях, когда вы самостоятельно разрабатываете различные части одной программы.

Предположим, вы хотите, чтобы ваше приложение с графическим пользовательским интерфейсом отображало диалоговое окно, в котором можно выбрать папку для сохранения документа. Отображаемые в диалоговом окне данные могут задаваться посредством аргументов, передаваемых вашим обработчикам событий. Однако для того, чтобы определить, как именно следует визуализировать информацию, диалоговое окно должно считывать также глобальное состояние, например информацию о предпочтениях пользователя.

Определим диалоговое окно, которое извлекает информацию о заданном по умолчанию месте сохранения файла из глобальных установок.

```
# dialog.py
import app
class Dialog(object):
    def __init__(self, save_dir):
        self.save_dir = save_dir
    # ...

save_dialog = Dialog(app.prefs.get('save_dir'))

def show():
    # ...
```

Проблема в том, что модуль `app`, который содержит объект `prefs`, импортирует также класс `dialog`, необходимый для отображения диалогового окна при запуске программы.

```
# app.py
import dialog

class Prefs(object):
    # ...
    def get(self, name):
        # ...

prefs = Prefs()
dialog.show()
```

Здесь налицо циклическая зависимость. Если вы попытаетесь использовать модуль `app` в своей основной программе, то при его импортировании возникнет исключение.

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    import app
  File "app.py", line 4, in <module>
    import dialog
```

```
  File "dialog.py", line 16, in <module>
    save_dialog = Dialog(app.prefs.get('save_dir'))
AttributeError: 'module' object has no attribute 'prefs'
```

Чтобы понять, что здесь происходит, вам необходимо более детально ознакомиться с работой механизма импорта Python. Ниже перечислены операции (начиная с наиболее низкоуровневых), которые выполняет Python при импортировании модуля:

1. выполняет поиск модуля в местах, указанных в `sys.path`;
2. загружает код из модуля и убеждается в его компиляции;
3. создает объект соответствующего модуля;
4. вставляет модуль в `sys.modules`;
5. выполняет код в объекте модуля для определения его содержимого.

Суть проблемы циклических зависимостей в том, что атрибуты модуля не определены до тех пор, пока не выполнится код для этих атрибутов (шаг 5). Однако модуль может быть загружен инструкцией `import` сразу же после того, как он вставлен в `sys.modules` (шаг 4).

В приведенном выше примере модуль `app` импортирует `dialog`, прежде чем что-либо определять, а затем модуль `dialog` импортирует модуль `app`. Поскольку `app` все еще не закончил выполнение (в данный момент он импортирует модуль `dialog`), он представляет собой всего лишь пустую оболочку (после шага 4). Генерируется исключение `AttributeError` (для модуля `dialog` на шаге 5), поскольку код, который определяет `prefs`, пока что не выполнялся (шаг 5 для `app` еще не завершился).

Наилучшее решение этой проблемы — выполнить рефакторинг кода, чтобы структура данных `prefs` оказалась в самом низу дерева зависимостей. Далее как `app`, так и `dialog` могут импортировать один и тот же модуль `utility` и избежать циклических зависимостей. Но такое четкое разделение не всегда возможно или же требует выполнения такого большого объема работы по рефакторингу кода, что затраченные усилия себя не оправдают.

Существуют еще три способа разрыва циклических зависимостей.

Изменение порядка импортирования модулей

Один из вышеупомянутых способов — это изменение порядка следования инструкций `import`. Например, если вы перенесете операцию импортирования модуля `dialog` в конец модуля `app`, чтобы дать возможность выполниться содержимому последнего, то исключение `AttributeError` не возникнет.

```
# app.py
class Prefs(object):
    # ...

prefs = Prefs()

import dialog # Перемещен
dialog.show()
```

Этот код работает, поскольку в случае поздней загрузки модуля `dialog` при попытке выполнения в нем рекурсивного импорта модуля `app` будет обнаружено, что объект `app.prefs` уже определен (шаг 5 для `app` в основном уже выполнен).

Несмотря на то что такой подход позволяет избежать исключений `AttributeError`, он не соответствует рекомендациям по стилю PEP 8 (см. раздел "Рекомендация 2. Руководствуйтесь правилами стилевого оформления программ, изложенными в документе PEP 8"). В соответствии с этими правилами инструкции `import` всегда должны располагаться в самом начале файлов Python. Благодаря этому тем, кто впервые читает ваш код, будет легче разобраться в зависимостях, существующих между модулями. Это также гарантирует, что любой модуль, от которого зависит ваш код, будет доступен всему коду вашего модуля.

Размещение инструкций `import` в конце файла повышает хрупкость кода и может увеличивать вероятность того, что даже небольшие изменения в порядке следования вашего кода сделают его полностью неработоспособным.

Таким образом, следует избегать изменения порядка импортирования модулей для разрешения проблем, связанных с циклическими зависимостями.

Импортирование, конфигурирование, выполнение

Другое решение проблемы циклического импорта заключается в сведении к минимуму вероятности возникновения побочных эффектов во время выполнения импорта. Старайтесь писать код так, чтобы ваши модули лишь определяли функции, классы и константы. Избегайте фактического выполнения каких-либо функций при импорте. Необходимо также предусматривать, чтобы каждый модуль предоставлял функцию `configure()`, которую вы вызываете, по завершении импорта всех других модулей. Назначением этой функции является подготовка состояния каждого модуля путем обращения к атрибутам других модулей. Вы выполняете функцию после того, как все модули были уже импортированы (завершен шаг 5), и поэтому все атрибуты должны быть определены.

Переопределим модуль dialog таким образом, чтобы доступ к объекту prefs осуществлялся лишь при вызове функции configure().

```
# dialog.py
import app

class Dialog(object):
    # ...

save_dialog = Dialog()

def show():
    # ...

def configure():
    save_dialog.save_dir = app.prefs.get('save_dir')
```

Мы также переопределим модуль app таким образом, чтобы во время импорта в нем не выполнялись никакие существенные действия.

```
# app.py
import dialog

class Prefs(object):
    # ...

prefs = Prefs()

def configure():
    # ...
```

Наконец, в модуле main имеются три различные фазы выполнения: импортирование всего необходимого, конфигурирование всего необходимого и выполнение первого действия.

```
# main.py
import app
import dialog

app.configure()
dialog.configure()

dialog.show()
```

Описанный подход годится для многих ситуаций и позволяет использовать такие шаблоны проектирования, как *внедрение зависимостей*. Но иногда трудно структурировать код так, чтобы настройку конфигурации модуля можно было выделить в отдельный этап. Кроме того, наличие

двух отдельных фаз в модуле может затруднить чтение кода, поскольку это отделяет определение объектов от настройки их конфигураций.

Динамический импорт

Третье — и часто простейшее — решение проблемы циклического импорта состоит в том, чтобы использовать инструкцию `import` в функции или методе. Это называется *динамическим импортом*, поскольку модуль импортируется во время выполнения программы, а не при первоначальном запуске программы и инициализации ее модулей.

Ниже мы переопределим модуль `dialog` так, чтобы использовать динамический импорт. Здесь вместо импортирования модуля `app` модулем `dialog` во время инициализации он импортируется функцией `dialog.show()` во время выполнения.

```
# dialog.py
class Dialog(object):
    # ...

save_dialog = Dialog()

def show():
    import app # Dynamic import
    save_dialog.save_dir = app.prefs.get('save_dir')
    # ...
```

Теперь модуль `app` может быть использован в том же виде, что и в исходном примере: модуль `dialog` импортируется в начале модуля `app`, а функция `dialog.show()` вызывается в конце.

```
# app.py
import dialog

class Prefs(object):
    # ...
prefs = Prefs()

dialog.show()
```

При таком подходе результирующий эффект на стадиях импортирования, конфигурирования и выполнения модулей аналогичен рассмотренному выше. Отличие в том, что это не требует внесения каких-либо структурных изменений, касающихся способа определения и импортирования модулей. Вы просто откладываете циклический импорт до тех пор, когда вам действительно не потребуется доступ к другому модулю. К этому моменту вы можете быть достаточно уверены в том,

что все другие модули уже инициализированы (шаг 5 уже завершен для всех модулей).

В целом лучше избегать динамического импорта. Накладными расходами операций импорта нельзя пренебрегать, особенно в случае интенсивных циклов. Кроме того, откладывая выполнение, динамический импорт может преподнести вам сюрпризы на этапе выполнения, такие как исключения `SyntaxError`, возникающие спустя длительное время после запуска программы (о том, как избежать подобных ситуаций, см. в разделе "Рекомендация 56. Тестируйте любой код с помощью модуля `unittest`"). Однако даже с учетом указанных недостатков данный способ часто оказывается лучше альтернативы в виде реструктуризации всей программы.

Что следует запомнить

- Циклические зависимости возникают в тех случаях, когда два модуля должны вызывать друг друга на стадии импорта. Это может привести к краху программы уже при запуске.
- Наилучшим способом разрыва циклических зависимостей является рефакторинг кода для вынесения взаимных зависимостей в отдельные модули в нижней части дерева зависимостей.
- Динамический импорт — простейшее решение, позволяющее устранить циклические зависимости между модулями с минимальными усилиями по выполнению рефакторинга кода.

Рекомендация 53. Используйте виртуальные среды для изолированных и воспроизводимых зависимостей

Процесс создания крупных программ повышенной сложности часто приводит к тому, что приходится опираться на различные пакеты, разработанные сообществом пользователей Python (см. раздел "Рекомендация 48. Знайте, где искать модули, разработанные сообществом Python"). При этом вы будете устанавливать такие пакеты, как `pytz`, `numpy` и многие другие, используя систему управления пакетами `pip`.

Проблема в том, что по умолчанию `pip` устанавливает новые пакеты в глобальных расположениях. В результате этого вновь установленные модули будут оказывать воздействие на все программы на языке Python, имеющиеся в вашей системе. Теоретически это не должно вызывать никаких проблем. Если вы устанавливаете пакет, но не импортируете его, то как он может влиять на ваши программы?

Проблема коренится в транзитивных зависимостях — пакетах, от которых зависят устанавливаемые вами пакеты. Например, вы можете увидеть зависимости пакета Sphinx после его установки, запросив об этом pip.

```
$ pip3 show Sphinx
---
Name: Sphinx
Version: 1.2.2
Location: /usr/local/lib/python3.4/site-packages
Requires: docutils, Jinja2, Pygments
```

Если вы установите другой пакет, например flask, то увидите, что он также зависит от пакета Jinja2.

```
$ pip3 show flask
---
Name: Flask
Version: 0.10.1
Location: /usr/local/lib/python3.4/site-packages
Requires: Werkzeug, Jinja2, itsdangerous
```

Причиной конфликта может стать возникновение расхождений между пакетами Sphinx и flask с течением времени. Возможно, сейчас им обоим требуется одна и та же версия пакета Jinja2, и все работает нормально. Но спустя полгода или год может быть выпущена новая версия Jinja2, которая заставляет пользователей библиотеки вносить изменения в свой код. Если вы обновите свою глобальную версию пакета Jinja2 с помощью команды pip install --upgrade, то может оказаться, что пакет Sphinx не сможет работать, тогда как на пакете flask это никак не отразится.

Такие нарушения работы кода возникают потому, что Python позволяет иметь в каждый момент времени лишь одну установленную глобальную версию модуля. Если один из установленных вами пакетов должен использовать новую версию, а другой — старую, то ваша система не будет работать так, как надо.

Подобные программные сбои могут случаться даже тогда, когда в процессе сопровождения кода делается все возможное для того, чтобы сохранить совместимость API при переходе от одной версии к другой (см. раздел “Рекомендация 50. Используйте пакеты для организации модулей и предоставления стабильных API”). Новые версии библиотеки могут незначительно изменять поведение модулей, на которое опирается код, использующий данный API. Пользователи системы могут обновить один пакет до новой версии, но не другие, ибо это может привести

к разрыву зависимостей. Таким образом, существует постоянная опасность того, что вы почувствуете, как земля уходит из-под ваших ног.

Эти трудности только увеличиваются в случае коллективной разработки, когда ваши коллеги работают на других компьютерах. Разумно предположить, что версии Python и глобальные пакеты, которые они установили на своих машинах, будут незначительно отличаться от ваших. Это может породить крайне неприятную ситуацию, когда кодовая база прекрасно работает на компьютере одного программиста, но отказывается работать на компьютере другого.

Ключом к решению любых проблем подобного рода является инструмент `pyvenv`, предоставляющий виртуальные среды. Начиная с версии Python 3.4 утилита командной строки `pyvenv` доступна по умолчанию вместе с установкой Python (она также доступна по команде `python -m venv`). Предыдущие версии Python требуют установки отдельного пакета (с помощью команды `pip install virtualenv`) и использования утилиты командной строки под названием `virtualenv`.

Средство `pyvenv` позволяет создавать изолированные версии среды Python. Используя `pyvenv`, вы сможете иметь одновременно много версий одного и того же пакета, установленных в одной системе, которые не конфликтуют между собой. Эта дает возможность работать со многими проектами и использовать множество различных инструментов на одном и том же компьютере.

Средство `pyvenv` обеспечивает это путем установки версий пакетов и их зависимостей в полностью независимые структуры каталогов. Это дает возможность воспроизводить ту среду Python, относительно которой у вас есть уверенность, что она будет работать с вашим кодом. Это надежный способ, позволяющий избежать неожиданной потери работоспособности кода.

Команда pyvenv

Рассмотрим пример, который поможет вам научиться эффективно использовать средство `pyvenv`. Прежде чем использовать этот инструмент, важно, чтобы вам был понятен смысл команды `python3`, которую вы вводите в командной строке своей системы. На моем компьютере `python3` располагается в каталоге `/usr/local/bin` и запускает версию 3.4.2 (см. раздел “Рекомендация 1. Следите за тем, какую версию Python вы используете”).

```
$ which python3
/usr/local/bin/python3
$ python3 --version
Python 3.4.2
```

Чтобы продемонстрировать настройку моей среды, я могу проверить, что выполнение команды для импортирования модуля pytz не приведет к ошибке. Это работает, поскольку пакет pytz уже установлен у меня в качестве глобального модуля.

```
$ python3 -c 'import pytz'
$
```

Далее я использую pyvenv для создания новой виртуальной среды под названием myproject.

Каждая виртуальная среда должна существовать в собственном уникальном каталоге. Результатом выполнения команды является создание дерева каталогов и файлов.

```
$ pyvenv /tmp/myproject
$ cd /tmp/myproject
$ ls
bin    include    lib    pyvenv.cfg
```

Чтобы начать использовать виртуальную среду, я применю команду source моей оболочки к сценарию bin/activate. Сценарий activate изменяет все переменные моей среды таким образом, чтобы они соответствовали виртуальной среде. Кроме того, он изменяет приглашение моей командной строки, включая в него имя виртуальной среды ('myproject'), чтобы было предельно ясно, в какой именно среде я работаю.

```
$ source bin/activate
(myproject)$
```

После того как среда активизирована, вы сможете увидеть, что путь к командной строке python3 переместился в каталог виртуальной среды.

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ ls -l /tmp/myproject/bin/python3
... -> /tmp/myproject/bin/python3.4
(myproject)$ ls -l /tmp/myproject/bin/python3.4
... -> /usr/local/bin/python3.4
```

Это гарантирует, что изменения во внешней системе не будут влиять на виртуальную среду. Даже если внешняя система обновляет свою версию по умолчанию, вызываемую командой python3, до версии 3.5, моя виртуальная среда будет по-прежнему явно указывать на версию 3.4.

Среда, которую я создал с помощью средства pyvenv, запускается без каких-либо установленных пакетов, не считая pip и setuptools. Попытка использовать пакет pytz, который был установлен в качестве гло-

бального модуля во внешней системе, окажется неудачной, поскольку он неизвестен виртуальной среде.

```
(myproject)$ python3 -c 'import pytz'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'pytz'
```

Я могу использовать систему управления пакетами `pip` для установки модуля `pytz` в мою виртуальную среду.

```
(myproject)$ pip3 install pytz
```

Как только модуль `pytz` установлен, я могу убедиться в том, что он работает, используя ту же тестовую команду импорта.

```
(myproject)$ python3 -c 'import pytz'
(myproject)$
```

Чтобы вернуться в свою заданную по умолчанию систему по окончании работы в виртуальной среде, используйте команду `deactivate`. Это восстановит вашу среду до заданной в системе по умолчанию, включая расположение средства командной строки `python3`.

```
(myproject)$ deactivate
$ which python3
/usr/local/bin/python3
```

Если вам когда-либо вновь понадобится работать в виртуальной среде, достаточно будет, как и прежде, выполнить команду `source bin/activate` в соответствующем каталоге.

Воспроизведение зависимостей

Организовав виртуальную среду, вы можете устанавливать в ней пакеты с помощью `pip` по мере необходимости. В конечном счете вы можете скопировать свою среду в другое место. Допустим, вы хотите воспроизвести свою среду разработки на производственном сервере. Или же вы хотите воспроизвести среду другого разработчика на своем компьютере, чтобы иметь возможность выполнять его код.

Средство `pyvenv` позволяет легко справляться с подобными ситуациями. Используя команду `pip freeze`, сохраните все явные зависимости своих пакетов в файле. В соответствии с принятым соглашением этот файл называется *requirements.txt*.

```
(myproject)$ pip3 freeze > requirements.txt
(myproject)$ cat requirements.txt
numpy==1.8.2
```

```
pytz==2014.4
requests==2.3.0
```

Допустим, что вы хотите установить другую виртуальную среду, которая соответствует среде `myproject`. Вы можете создать новый каталог и активизировать среду, как мы делали до этого, используя средства `pyvenv` и `activate`.

```
$ pyvenv /tmp/otherproject
$ cd /tmp/otherproject
$ source bin/activate
(otherproject)$
```

В новой среде не будут установлены никакие дополнительные пакеты.

```
(otherproject)$ pip3 list
pip (1.5.6)
setuptools (2.1)
```

Можно установить все пакеты из первой среды, выполнив команду `pip install` для файла *requirements.txt*, который был сгенерирован с помощью команды `pip freeze`.

```
(otherproject)$ pip3 install -r /tmp/myproject/requirements.txt
```

Эта команда немного потрудится, пока не извлечет и не установит все пакеты, необходимые для воспроизведения первой среды. Как только эта работа будет выполнена, перечисление набора установленных пакетов во второй виртуальной среде приведет к тому же списку зависимостей, что и в случае первой виртуальной среды.

```
(otherproject)$ pip list
numpy (1.8.2)
pip (1.5.6)
pytz (2014.4)
requests (2.3.0)
setuptools (2.1)
```

Использование файла *requirements.txt* идеально подходит для организации коллективной работы посредством системы управления версиями. Вы можете принять внесенные в код изменения одновременно с обновлением списка пакетных зависимостей с гарантией того, что это будет сделано синхронно.

Неприятным моментом при работе с виртуальными средами является то, что их перемещение разрушает все связи, поскольку все пути (например, путь к `python3`) жестко закодированы в каталоге установки среды. Но это не имеет значения. Единственная цель использования виртуальных сред состоит в том, чтобы упростить воспроизведение одной

и той же установки. Вместо перемещения каталога виртуальной среды достаточно вывести в файл дамп старого каталога, создать новый каталог в другом месте и переустановить все из файла *requirements.txt*.

Что следует запомнить

- Виртуальные среды позволяют использовать средство `pip` для установки нескольких версий одного и того же пакета на одной и той же машине так, что между ними не будут возникать конфликты.
- Виртуальные среды создаются с помощью команды `pyvenv`, активизируются с помощью команды `source bin/activate` и отключаются с помощью команды `deactivate`.
- Вы можете вывести дамп всех требований к среде с помощью команды `pip freeze`. Для воспроизведения среды следует предоставить команде `pip install -r` файл *requirements.txt*.
- В версиях Python ниже 3.4 средство `pyvenv` должно загружаться и устанавливаться отдельно. Средство командной строки называется не `pyvenv`, а `virtualenv`.