

Recipes for Mastering Python 3

3rd Edition



Python Cookbook

O'REILLY®

David Beazley & Brian K. Jones

Этот [перевод](#) выполнен в целях самообучения. Он не предназначен для издания, продажи и какого-либо иного коммерческого использования. Пожалуйста, купите бумажное или электронное издание на английском языке [здесь](#).

Содержание

1. Структуры данных и алгоритмы
 - 1.1. Распаковка последовательности в отдельные переменные

- 1.2. Распаковка элементов из последовательностей произвольной длины
- 1.3. Оставляем N последних элементов
- 1.4. Поиск N максимальных и минимальных элементов
- 1.5. Реализация очереди с приоритетом
- 1.6. Отображение ключей на несколько значений в словаре
- 1.7. Поддержание порядка в словарях
- 1.8. Вычисления на словарях
- 1.9. Поиск общих элементов в двух словарях
- 1.10. Удаление дубликатов из последовательности с сохранением порядка элементов
- 1.11. Присваивание имён срезам
- 1.12. Определение наиболее часто встречающихся элементов в последовательности
- 1.13. Сортировка списка словарей по общему ключу
- 1.14. Сортировка объектов, не поддерживающих сравнение
- 1.15. Группирование записей на основе полей
- 1.16. Фильтрование элементов последовательности
- 1.17. Извлечение подмножества из словаря
- 1.18. Отображение имен на последовательность элементов
- 1.19. Одновременное преобразование и сокращение (свёртка) данных
- 1.20. Объединение нескольких отображений в одно

2. Строки и текст

- 2.1. Разрезание строк, разделенных различными разделителями
- 2.2. Поиск текста в начале и в конце строки
- 2.3. Поиск строк с использованием масок оболочки (shell)
- 2.4. Поиск совпадений и поиск текстовых паттернов
- 2.5. Поиск и замена текста
- 2.6. Поиск и замена текста без учета регистра
- 2.7. Определение регулярных выражений для поиска кратчайшего совпадения
- 2.8. Написание регулярного выражения для многострочных шаблонов
- 2.9. Приведение текста в Unicode к стандартному представлению (нормализация)
- 2.10. Использование символов Unicode в регулярных выражениях
- 2.11. Убиение нежелательных символов из строк
- 2.12. Чистка строк

- 2.13. Выравнивание текстовых строк
- 2.14. Объединение и конкатенация строк
- 2.15. Интерполяция переменных в строках
- 2.16. Разбивка текста на фиксированное количество колонок
- 2.17. Работа с HTMLи XML-сущностями в тексте
- 2.18. Токенизация текста
- 2.19. Написание простого парсера на основе метода рекурсивного спуска
- 2.20. Выполнение текстовых операций над байтовыми строками

3. Числа, даты и время

- 3.1. Округление числовых значений
- 3.2. Выполнение точных десятичных вычислений
- 3.3. Форматирование чисел для вывода
- 3.4. Работа с бинарными, восьмеричными и шестнадцатеричными целыми числами
- 3.5. Упаковка и распаковка больших целых чисел из байтовых строк
- 3.6. Вычисления с комплексными числами
- 3.7. Работа с бесконечными значениями и NaN
- 3.8. Вычисления с дробями
- 3.9. Вычисления на больших массивах чисел
- 3.10. Вычисления с матрицами и линейная алгебра
- 3.11. Случайный выбор
- 3.12. Перевод дней в секунды и другие базовые методы конвертации времени
- 3.13. Определение даты последней пятницы
- 3.14. Поиск диапазона дат для текущего месяца
- 3.15. Конвертирование строк в даты и время
- 3.16. Манипулирование датами с учётом таймзон

4. Итераторы и генераторы

- 4.1. Ручное прохождение по итератору
- 4.2. Делегирование итерации
- 4.3. Создание новых итерационных паттернов с помощью генераторов
- 4.4. Реализация протокола итератора
- 4.5. Итерирование в обратном порядке
- 4.6. Определение генератора с дополнительным состоянием
- 4.7. Получение среза итератора
- 4.8. Пропуск первой части итерируемого объекта
- 4.9. Итерирование по всем возможным комбинациям и

перестановкам

- 4.10. Итерирование по парам «индекс-значение» последовательности
- 4.11. Одновременное итерирование по нескольким последовательностям
- 4.12. Интериорирования по элементам, находящимся в отдельных контейнерах
- 4.13. Создание каналов для обработки данных
- 4.14. Превращение вложенной последовательности в плоскую
- 4.15. Последовательное итерирование по слитым отсортированным итерируемым объектам
- 4.16. Замена бесконечных циклов `while` итератором

5. Файлы и ввод-вывод

- 5.1. Чтение и запись текстовых данных
- 5.2. Перенаправление вывода в файл
- 5.3. Вывод с другим разделителем или символом конца строки
- 5.4. Чтение и запись бинарных данных
- 5.5. Запись в файл, которого ещё нет
- 5.6. Выполнение операций ввода-вывода над строками
- 5.7. Чтение и запись сжатых файлов с данными
- 5.8. Итерирование по записям фиксированного размера
- 5.9. Чтение бинарных данных в изменяемый (мутабельный) буфер
- 5.10. Отображаемые в память бинарные файлы
- 5.11. Манипулирование путями к файлам
- 5.12. Проверка существования файла
- 5.13. Получение содержимого каталога
- 5.14. Обход кодировки имен файлов
- 5.15. Вывод «плохих» имён файлов
- 5.16. Добавление или изменение кодировки уже открытого файла
- 5.17. Запись байтов в текстовый файл
- 5.18. Оборачивание существующего дескриптора файла для использования в качестве объекта файла
- 5.19. Создание временных файлов и каталогов
- 5.20. Работа с последовательными портами
- 5.21. Сериализация объектов Python

6. Кодирование и обработка данных

- 6.1. Чтение и запись данных в формате CSV
- 6.2. Чтение и запись в формате JSON
- 6.3. Парсинг простых XML-данных
- 6.4. Инкрементальный парсинг очень больших XML-файлов

- 6.5. Преобразование словарей в XML
- 6.6. Парсинг, изменение и перезапись XML
- 6.7. Парсинг XML-документов с пространствами имён
- 6.8. Взаимодействие с реляционной базой данных
- 6.9. Декодирование и кодирование шестнадцатеричных цифр
- 6.10. Кодирование и декодирование в Base64
- 6.11. Чтение и запись бинарных массивов структур
- 6.12. Чтение вложенных и различных по размеру бинарных структур
- 6.13. Суммирование данных и обсчёт статистики

7. Функции

- 7.1. Определение функций, принимающих любое количество аргументов
- 7.2. Определение функций, принимающих только именованные аргументы
- 7.3. Прикрепление информационных метаданных к аргументам функций
- 7.4. Возвращение функцией нескольких значений
- 7.5. Определение функций с аргументами по умолчанию
- 7.6. Определение анонимных функций или функций в строке (инлайновых)
- 7.7. Захват переменных в анонимных функциях
- 7.8. Заставляем вызываемый объект с N аргументами работать так же, как вызываемый объект с меньшим количеством аргументов
- 7.9. Замена классов с одним методом функциями
- 7.10. Передача дополнительного состояния с функциями обратного вызова
- 7.11. Внутристрочные функции обратного вызова
- 7.12. Доступ к переменным, определенным внутри замыкания

8. Классы и объекты

- 8.1. Изменение строкового представления экземпляров
- 8.2. Настройка строкового форматирования
- 8.3. Создание объектов, поддерживающих протокол менеджера контекста
- 8.4. Экономия памяти при создании большого количества экземпляров
- 8.5. Инкапсуляция имён в классе
- 8.6. Создание управляемых атрибутов
- 8.7. Вызов метода родительского класса
- 8.8. Расширение свойства в подклассе

- 8.9. Создание нового типа атрибута класса или экземпляра
- 8.10. Использование лениво вычисляемых свойств
- 8.11. Упрощение инициализации структур данных
- 8.12. Определение интерфейса или абстрактного базового класса
- 8.13. Реализации модели данных или системы типов
- 8.14. Реализация собственных контейнеров
- 8.15. Делегирование доступа к атрибуту
- 8.16. Определение более одного конструктора в классе
- 8.17. Создание экземпляра без вызова *init*
- 8.18. Расширение классов с помощью миксин (примесей)
- 8.19. Реализация объектов с состоянием или конечных автоматов
- 8.20. Вызов метода объекта с передачей имени метода в строке
- 8.21. Реализация шаблона проектирования «посетитель»
- 8.22. Реализация шаблона «посетитель» без рекурсии
- 8.23. Управление памятью в циклических структурах данных
- 8.24. Заставляем классы поддерживать операции сравнения
- 8.25. Создание кэшированных экземпляров

9. Метапрограммирование

- 9.1. Создание обёртки для функции
- 9.2. Сохранение метаданных функции при написании декораторов
- 9.3. Снятие («разворачивание») декоратора
- 9.4. Определение декоратора, принимающего аргументы
- 9.5. Определение декоратора с настраиваемыми пользователем атрибутами
- 9.6. Определение декоратора, принимающего необязательный аргумент
- 9.7. Принудительная проверка типов в функции с использованием декоратора
- 9.8. Определение декораторов как части класса
- 9.9. Определение декораторов как классов
- 9.10. Применение декораторов к методам класса и статическим методам
- 9.11. Написание декораторов, которые добавляют аргументы обёрнутым функциям
- 9.12. Использование декораторов как патчей определений классов
- 9.13. Использование метакласса для управления созданием экземпляров
- 9.14. Захват порядка определения атрибутов класса
- 9.15. Определение метакласса, принимающего необязательные

аргументы

- 9.16. Принудительная установка аргументной сигнатуры при использовании *args и **kwargs
- 9.17. Принуждение к использованию соглашений о кодировании в классах
- 9.18. Программное определение классов
- 9.19. Инициализация членов класса во время определения
- 9.20. Реализация множественной диспетчеризации с помощью аннотаций функций
- 9.21. Избежание повторяющихся методов свойств
- 9.22. Лёгкий способ определения менеджеров контекста
- 9.23. Выполнение кода с локальными побочными эффектами
- 9.24. Парсинг и анализ исходного кода Python
- 9.25. Дизассемблирование байт-кода Python

10. Модули и пакеты

- 10.1. Создание иерархического пакета модулей
- 10.2. Контроль импортирования
- 10.3. Импортирование подмодулей пакета с использованием относительных имён
- 10.4. Разделение модуля на несколько файлов
- 10.5. Заставляем отдельные каталоги с кодом импортироваться под общим пространством имён
- 10.6. Перезагрузка модулей
- 10.7. Создание каталога или zip-архива, запускаемых как единственный скрипт
- 10.8. Чтение файлов с данными внутри пакета
- 10.9. Добавление каталогов в sys.path
- 10.10. Импортирование модулей с использованием передаваемого в форме строки имени
- 10.11. Загрузка модулей с удалённого компьютера с использованием хуков импортирования
- 10.12. Применение к модулям патчей во время импортирования
- 10.13. Установка пакетов «только для себя»
- 10.14. Создание нового окружения Python
- 10.15. Распространение пакетов

11. Сети и веб-программирование

- 11.1. Взаимодействие с HTTP-сервисами в роли клиента
- 11.2. Создание TCP-сервера
- 11.3. Создание UDP-сервера

- 11.4. Генерация диапазона IP-адресов из CIDR-адреса
- 11.5. Создание простого REST-интерфейса
- 11.6. Реализация простого удалённого вызова процедуры через XML-RPC
- 11.7. Простое взаимодействие между интерпретаторами
- 11.8. Реализация вызовов удалённых процедур
- 11.9. Простая аутентификация клиентов
- 11.10. Добавление SSL в сетевые сервисы
- 11.11. Передача файловых дескрипторов сокетов между процессами
- 11.12. Разбираемся с вводом-выводом, управляемым событиями (event-driven I/O)
- 11.13. Отсылка и получение больших массивов

12. Конкурентное программирование

- 12.1. Запуск и остановка потоков
- 12.2. Как узнать, стартовал ли поток
- 12.3. Коммуникация между потоками
- 12.4. Блокировка критически важных участков
- 12.5. Блокировка с избежанием дедлока
- 12.6. Хранение «потокоспецифичного» состояния
- 12.7. Создание пула потоков
- 12.8. Простое параллельное программирование
- 12.9. Разбираемся с GIL (и перестаём волноваться по этому поводу)
- 12.10. Определение акторной задачи
- 12.11. Реализация системы сообщений «опубликовать/подписаться» (pub/sub)
- 12.12. Использование генераторов в качестве альтернативы потокам
- 12.13. Опрашивание многопоточных очередей
- 12.14. Запуск процесса-демона на Unix

13. Полезные скрипты и системное администрирование

- 13.1. Скрипты, принимающие ввод через перенаправление, каналы или файлы
- 13.2. Завершение программы с выводом сообщения об ошибке
- 13.3. Парсинг аргументов командной строки
- 13.4. Запрос пароля во время выполнения
- 13.5. Получение размера окна терминала
- 13.6. Выполнение внешней команды и получение её вывода
- 13.7. Копирование или перемещение файлов и каталогов
- 13.8. Создание и распаковка архивов
- 13.9. Поиск файлов по имени

- 13.10. Чтение конфигурационных файлов
- 13.11. Добавление логирования в простые скрипты
- 13.12. Добавление логирования в библиотеки
- 13.13. Создание таймера-секундомера
- 13.14. Установка лимитов на использование памяти и CPU
- 13.15. Запуск браузера

14. Тестирование, отладка и исключения

- 14.1. Тестирование отправки вывода в stdout
- 14.2. Патчинг объектов в юнит-тестах
- 14.3. Проверка вызывающих исключения условий в рамках юнит-тестов
- 14.4. Логирование вывода теста в файл
- 14.5. Пропуск или ожидание провалов тестов
- 14.6. Обработка множественных исключений
- 14.7. Ловим все исключения
- 14.8. Создание собственных исключений
- 14.9. Возбуждение исключения в ответ на другое исключение
- 14.10. Повторное возбуждение последнего исключения
- 14.11. Вывод предупреждающих сообщений
- 14.12. Отладка базовых падений программ
- 14.13. Профилирование и замеры времени выполнения вашей программы
- 14.14. Заставляем ваши программы выполнять быстрее

15. Расширения на языке C

- 15.1. Доступ к коду на C с использованием ctypes
- 15.2. Написание простого модуля расширения на C
- 15.3. Написание функции расширения для работы с массивами
- 15.4. Управление непрозрачными указателями в модулях расширения на C
- 15.5. Определение и экспорт C API из модулей расширения
- 15.6. Вызываем Python из C
- 15.7. Освобождение GIL в расширениях на C
- 15.8. Смешивание потоков из C и Python
- 15.9. Оборачивание кода на C в Swig
- 15.10. Оборачивание существующего кода на C в Cython
- 15.11. Использование Cython для высокопроизводительных операций над массивами
- 15.12. Превращение указателя на функцию в вызываемый объект
- 15.13. Передача NULL-терминированных строк библиотекам на C

- 15.14. Передача строк Unicode в библиотеки на C
- 15.15. Преобразование строк C в Python
- 15.16. Работа со строками C в сомнительной кодировке
- 15.17. Передача имён файлов в расширения на C
- 15.18. Передача открытых файлов в расширения на C
- 15.19. Чтение файлоподобных объектов из C
- 15.20. Потребление итерируемого объекта из C
- 15.21. Диагностика ошибок сегментации

1. Структуры данных и алгоритмы

Python предоставляет широкий спектр встроенных структур данных, таких как списки, множества и словари. Использовать эти структуры по большей части просто. Однако часто возникают вопросы, касающиеся поиска, сортировки, изменения порядка элементов и фильтрования. Цель этой главы – обсудить распространённые структуры данных и алгоритмы. Также будет дано введение в структуры данных из модуля *collections*.

1.1. Распаковка последовательности в отдельные переменные

Задача

У вас есть кортеж из N элементов или последовательность, которую вы хотите распаковать в коллекцию из N переменных.

Решение

Любая последовательность (или итерируемый объект) может быть распакована в переменные с помощью простого присваивания. Единственное обязательное условие заключается в том, чтобы количество и структура переменных совпадали с таковыми у последовательности. Например:

```
>>> p = (4, 5)
>>> x, y = p
```

```
>>> x
4
>>> y
5
>>>

>>> data = ['ACME', 50, 91.1, (2012, 12, 21)]
>>> name, shares, price, date = data
>>> name
'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>
```

При несовпадении количества элементов вы получите ошибку. Например:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

Обсуждение

Распаковка работает с любым итерируемым объектом, а не только с кортежами и списками. Это строки, файлы, итераторы и генераторы. Например:

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
```

```
>>>
```

При распаковке вы иногда можете захотеть отбраковать некоторые значения. Специального синтаксиса для этого в Python нет, но вы можете назначить переменные, которые потом отбросите. Например:

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

Но убедитесь, что вы уже не использовали где-то эту переменную.

1.2. Распаковка элементов из последовательностей произвольной длины

Задача

Вам нужно распаковать N элементов из итерируемого объекта, но этот объект может содержать больше N элементов, что вызывает исключение “too many values to unpack” («слишком много значений для распаковки»).

Решение

Для решения этой задачи можно использовать «выражения со звёздочкой». Предположим, например, что вы ведете учебный курс. В конце семестра вы решаете, что не будете принимать во внимание оценки за первое и последнее домашние задания, а по остальным оценкам посчитаете среднее значение. Если у вас было четыре задания, то можно просто распаковать все четыре. Но что делать, если их 24? Выражение со звёздочкой позволит легко решить эту задачу:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

Рассмотрим еще один пример: предположим, что у вас есть записи о пользователях, которые состоят из имени и адреса электронной почты, за которыми следует произвольное количество телефонных номеров. Вы можете распаковать записи так:

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```



Стоит отметить, что переменная *phone_numbers* всегда будет списком — несмотря на то, сколько телефонных номеров распаковано (даже если и ни одного). Любому коду, который будет использовать переменную *phone_numbers*, не нужно учитывать возможность того, что в ней будет не список (и производить дополнительные проверки на предмет этого).

Переменная со звёздочкой также может быть первой в списке. Например, у вас есть последовательность значений, представляющая продажи вашей компании за последние восемь кварталов. Если вы хотите посмотреть, как последний квартал соотносится со средним значением по первым семи, вы можете сделать так:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

Интерпретатор Python выдаст:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

Обсуждение

Расширенная распаковка отлично подходит для распаковки итерируемых

объектов неизвестной или произвольной длины. Часто эти объекты имеют некоторые известные элементы или шаблоны (например, «всё, что после элемента 1, является телефонным номером»). Распаковка со звёздочкой позволяет программисту легко использовать эти шаблоны — вместо того, чтобы выполнять акробатические трюки для извлечения нужных элементов из итерируемого объекта.

Стоит отметить, что синтаксис звёздочки может быть особенно полезен при итерировании по последовательности кортежей переменной длины. Например, возможна такая последовательность кортежей с тегами:

```
records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

Распаковка со звёздочкой также может быть полезна в комбинации с операциями обработки строк, такими как разрезание. Например:

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

Иногда вам может быть нужно распаковать значения и отбросить их. Вы не можете просто определить голую * при распаковке, но можно использовать обычное для отбрасывания имя переменной, такое как _ или ign (ignored).

Например:

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_ , (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

Есть некоторое сходство между распаковкой со звёздочкой и обработкой списков в функциональных языках. Например, если у вас есть список, то вы можете легко разделить его на «хвост» и «голову»:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

Можно представить себе функцию, которая произведет такое разрезание с помощью хитрого рекурсивного алгоритма. Например:

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

Однако вам следует знать, что рекурсия не относится к числу сильных сторон Python из-за внутреннего лимита на рекурсию. Поэтому последний пример на практике оказывается просто любопытным предметом для размышлений.

1.3. Оставляем N последних элементов

Задача

Вы хотите хранить ограниченную историю из нескольких последних

элементов, полученных в ходе итерации или какого-то другого процесса обработки данных.

Решение

Задача хранения ограниченной истории — отличный повод применить `collections.deque`. Например, следующий фрагмент кода производит простое сопоставление текста с последовательностью строк, а при совпадении выдает совпавшие строки вместе с N предыдущими строками контекста:

```
from collections import deque
def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
            previous_lines.append(line)

# Пример использования
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end=' ')
            print(line, end=' ')
            print('-'*20)
```

Обсуждение

При написании программы для поиска элементов обычно используют функцию-генератор, содержащую `yield` (как и показано в вышеприведенном примере). Это отделяет процесс поиска от кода, который использует результаты. Если вы новичок в обращении с генераторами, см. [Рецепт 4.3](#).

Использование `deque(maxlen=N)` создает очередь фиксированной длины. Когда новые элементы добавлены и очередь заполнена, самый старый элемент автоматически удаляется. Пример:

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
```

```
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

Хотя вы можете вручную производить такие операции над списком (то есть добавление в конец, удаление и т.п.), очередь элегантнее и работает намного быстрее.

Обобщим: дека может быть использована в любом случае, когда вам нужна простая очередь. Если вы не зададите максимальную длину, вы получите бесконечную очередь, которая позволит вам добавлять и удалять элементы с обоих концов. Например:

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

Добавление или удаление элементов в любой из концов очереди имеет сложность $O(1)$. А вот добавление или удаление элемента в начале списка имеет сложность $O(N)$.

1.4. Поиск N максимальных и минимальных элементов

Задача

Вы хотите создать список N максимальных или минимальных элементов коллекции.

Решение

У модуля `heapq` есть две функции, `nlargest()` и `nsmallest()`, которые делают именно то, что вам нужно. Например:

```
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Выведет [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Выведет [-4, 1, 2]
```

Обе функции также принимают параметр `key`, который позволяет использовать их с более сложными структурами данных. Например:

```
portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]
cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

Обсуждение

Если вы ищете N наименьших или N наибольших элементов, причем N невелико по сравнению с общим размером коллекции, эти функции покажут великолепную производительность. «Под капотом» они начинают работу с конвертирования данных в список, где данные упорядочены, как в куче.

Например:

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

Самое важная возможность кучи состоит в том, что `heap[0]` всегда будет наименьшим элементом. Кроме того, последующие элементы могут быть

легко найдены с помощью метода `heappq.heappop()`, который удаляет первый элемент и заменяет его следующим наименьшим элементом (это требует $O(\log N)$ операций, где N – размер кучи). Например, чтобы найти три наименьших элемента, вы могли бы сделать так:

```
>>> heappq.heappop(heap)
-4
>>> heappq.heappop(heap)
1
>>> heappq.heappop(heap)
2
```

Функции `nlargest()` и `nsmallest()` лучше всего подходят, если вы пытаетесь найти относительно небольшое количество элементов. Если же вы просто хотите найти один наибольший или наименьший элемент ($N = 1$), функции `min()` и `max()` будут быстрее. Похожим образом, если N сопоставимо с размером самой коллекции, обычно будет быстрее отсортировать их и взять срез (то есть выполнить `sorted(items)[:N]` или `sorted(items)[-N:]`). Стоит отметить, что реализация `nlargest()` и `nsmallest()` в модуле `heappq` работает гибко и выполняет некоторые из этих оптимизаций самостоятельно (например, использует сортировку, если размер N близок к размеру входных данных).

Хотя использовать этот рецепт необязательно, реализация кучи интересна и заслуживает изучения. Информацию о ней можно найти в любой приличной книге по алгоритмам и структурам данных. В документации модуля `heappq` также обсуждаются детали внутренней реализации.

1.5. Реализация очереди с приоритетом

Задача

Вы хотите реализовать очередь, которая сортирует элементы по заданному приоритету и всегда возвращает элемент с наивысшим приоритетом при каждой операции получения (удаления) элемента.

Решение

Приведенный ниже класс использует модуль `heappq` для реализации простой очереди с приоритетом.

```

import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]

```

А вот пример использования:

```

>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>

```

Первая операция `pop()` возвращает элемент с наивысшим приоритетом.

Также заметьте, что два элемента с одинаковым приоритетом (`foo` и `grok`) были возвращены в том же порядке, в каком они были помещены в очередь.

Обсуждение

Суть этого рецепта — использование модуля `heapq`. Функции `heapq.heappush()` и `heapq.heappop()` вставляют и удаляют элементы из `list_queue` таким образом, что первый элемент в списке имеет наименьший

приоритет (как обсуждалось в [рецепте 1.4.](#)). Метод `heappop()` всегда возвращает «наименьший» элемент, что является ключом к тому, чтобы заставить очередь удалять правильные элементы. Кроме того, так как операции вталкивания и снятия имеют сложность $O(\log N)$, где N — число элементов в куче, то они вполне эффективны даже для весьма больших значений N .

В этом рецепте очередь состоит из кортежей формата `(-priority, index, item)`. Значение приоритета сделано отрицательным, чтобы заставить очередь сортировать элементы от наибольшего к наименьшему приоритету. Это противоположно обычному порядку сортировки кучи (от наименьшего к наибольшему значению).

Роль переменной `index` заключается в установлении правильного порядка элементов с одинаковым приоритетом. Поддержание постоянно увеличивающегося индекса позволяет сортировать элементы в соответствии с порядком, в каком они были вставлены. Однако индекс также играет важную роль в выполнении операций сравнения при работе с элементами с одинаковыми значениями приоритета.

Если остановиться на этом подробнее, то отметим, что экземпляры класса `Item` не могут быть упорядочены. Например:

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Если вы создаете кортежи `(priority, item)`, то их можно сравнивать до тех пор, пока приоритеты различны. Однако же если сравниваются два кортежа с равными приоритетами, то сравнение не может быть проведено (как и ранее). Например:

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Вводя дополнительный индекс и создавая кортежи (*priority*, *index*, *item*), вы полностью обходите эту проблему, поскольку два кортежа никогда не будут иметь одинаковые значения переменной *index* (и Python никогда не будет сравнивать остальные значения в кортежах, если результат сравнения уже определен):

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

Если вы хотите использовать эту очередь для коммуникации между потоками (treadами), вы должны добавить правильную блокировку и передачу сигналов (см. [рецепт 12.3.](#))

[Документация](#) модуля *heapq* содержит дополнительные примеры и обсуждения теории и реализации куч.

1.6. Отображение ключей на несколько значений в словаре

Задача

Вы хотите создать словарь, который отображает ключи на более чем одно значение (так называемый «мультисловарь», *multidict*).

Решение

Словарь — это отображение, где каждый ключ отображен на единственное значение. Если вы хотите отобразить ключи на множественные значения, вам нужно хранить этим множественные значения в контейнере, таком как список или множество. Например, вы можете создавать такие словари:

```
d = {  
    'a' : [1, 2, 3],  
    'b' : [4, 5]  
}  
  
e = {  
    'a' : {1, 2, 3},  
    'b' : {4, 5}  
}
```

Выбор того, использовать или не использовать списки или множества, зависит от того, как будет использован мультисловарь. Применяйте список, если вы хотите сохранить порядок, в котором добавлены элементы. Применяйте множество, если вы хотите устраниить дубликаты (и при этом не беспокоитесь о порядке элементов).

Чтобы легко создавать такие словари, вы можете использовать *defaultdict* из модуля *collections*. Особенность *defaultdict* заключается в автоматической инициализации первого значения, так что вы можете сосредоточиться на добавлении элементов. Например:

```
from collections import defaultdict  
  
d = defaultdict(list)  
d['a'].append(1)  
d['a'].append(2)  
d['b'].append(4)  
...  
  
d = defaultdict(set)  
d['a'].add(1)  
d['a'].add(2)  
d['b'].add(4)  
...
```

Одно предупреждение: *defaultdict* автоматически создаст записи словаря для ключей, к которым позже будет осуществлен доступ (даже если их в данный момент в словаре нет). Если такое поведение нежелательно, вы можете использовать *setdefault()* на обычном словаре. Например:

```
d = {} # Обычный словарь  
d.setdefault('a', []).append(1)  
d.setdefault('a', []).append(2)  
d.setdefault('b', []).append(4)  
...
```

Однако многие программисты находят `setdefault()` несколько неестественным — и это если не учитывать тот факт, что он всегда создает новый экземпляр первоначального значения при каждом вызове (в примере это пустой список `[]`).

Обсуждение

Конструирование словарей с множественными значениями не является чем-то сложным. Однако инициализация первого значения может быть запутанной, если вы пытаетесь сделать это самостоятельно. Например, вы можете написать что-то такое:

```
d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
        d[key].append(value)
```

Использование `defaultdict` приводит к намного более чистому коду:

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

Этот рецепт связан с проблемой группировки записей в задачах обработки данных. Посмотрите, например, рецепт [1.15](#).

1.7. Поддержание порядка в словарях

Задача

Вы хотите создать словарь, который позволит контролировать порядок элементов при итерировании по нему или при сериализации.

Решение

Чтобы контролировать порядок элементов в словаре, вы можете использовать `OrderedDict` из модуля `collections`. При итерировании он в точности сохраняет изначальный порядок добавления данных. Например:

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Выведет "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

OrderedDict особенно полезен, когда вы хотите создать отображение, которое в дальнейшем собираетесь сериализовать или закодировать в другой формат. Например, если вы хотите строго контролировать порядок полей, выводимых в JSON, вам нужно просто создать *OrderedDict* с нужными данными:

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

Обсуждение

OrderedDict внутри себя поддерживает двусвязный список, который упорядочивает ключи в соответствии с порядком добавления. Когда новый элемент вставляется впервые, он помещается в конец этого списка. Последующее связывание значения с существующим ключом не изменяет порядок.

Заметьте, что размер *OrderedDict* более чем в два раза превышает размер обычного словаря из-за содержащегося внутри дополнительного списка. А если вы собираетесь создать структуру данных, в которой будет большое число экземпляров *OrderedDict* (например, вы хотите прочитать 100 000 строк CSV-файла в список экземпляров *OrderedDict*), вам стоит изучить требования вашего приложения, чтобы решить, перевесят ли преимущества использования *OrderedDict* оверхед на дополнительную память.

1.8. Вычисления на словарях

Задача

Вы хотите проводить различные вычисления (например, поиск минимального и максимального значения, сортировку) на словаре с данными.

Решение

Рассмотрим словарь, который отображает тикеры (идентификаторы акций на бирже) на цены:

```
prices = {  
    'ACME': 45.23,  
    'AAPL': 612.78,  
    'IBM': 205.55,  
    'HPQ': 37.20,  
    'FB': 10.75  
}
```

Чтобы выполнить вычисления на содержимом словаря, часто бывает полезно обратить ключи и значения, используя функцию `zip()`. Например, вот так можно найти минимальную и максимальную цену, а также соответствующий тикер:

```
min_price = min(zip(prices.values(), prices.keys()))  
# min_price - (10.75, 'FB')  
  
max_price = max(zip(prices.values(), prices.keys()))  
# max_price - (612.78, 'AAPL')
```

Похожим образом для ранжирования данных можно использовать `zip()` с `sorted()`, как показано ниже:

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))  
# prices_sorted - [(10.75, 'FB'), (37.2, 'HPQ'),  
#                   (45.23, 'ACME'), (205.55, 'IBM'),  
#                   (612.78, 'AAPL')]
```

Когда вы производите эти вычисления, обратите внимание, что `zip()` создает итератор, по которому можно пройти только один раз. Например, следующий фрагмент кода — неправильный:

```
prices_and_names = zip(prices.values(), prices.keys())  
print(min(prices_and_names)) # OK
```

```
print(max(prices_and_names)) # ValueError: max() arg is an empty sequence
```

Обсуждение

Если вы попытаетесь выполнить обычные функции обработки данных на словаре, то вы обнаружите, что они обрабатывают только ключи, но не значения. Например:

```
min(prices) # Вернёт 'AAPL'  
max(prices) # Вернёт 'IBM'
```

Вероятно, это не то, чего вы хотели добиться, поскольку вы пытались выполнить вычисления с использованием значений словаря. Вы можете попробовать исправить это, используя метод словаря `values()`:

```
min(prices.values()) # Вернёт 10.75  
max(prices.values()) # Вернёт 612.78
```

К несчастью, в большинстве случаев это не то, чего вы хотите. Например, вам нужно знать соответствующие ключи (т.е., понять, у каких акций самая низкая цена).

Вы можете получить ключ, соответствующий минимальному или максимальному значению, если вы передадите функцию в функции `min()` и `max()`. Например:

```
min(prices, key=lambda k: prices[k]) # Вернёт 'FB'  
max(prices, key=lambda k: prices[k]) # Вернёт 'AAPL'
```

Однако чтобы получить минимальное значение, вам потребуется дополнительное обращение. Например:

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

Решение с использованием функции `zip()` решает задачу путем «обращения» словаря в последовательность пар (`value, key`). Когда выполняется сравнение таких кортежей, элемент `value` сравнивается первым, а `key` — следующим. Это дает вам то самое поведение, которое вы хотите, а также позволяет проводить обработки и сортировку словаря с использованием единственного выражения.

Стоит отметить, что в вычислениях с использованием пар *(value, key)* будет использован *key*, чтобы определить результат в экземплярах, где множественные записи имеют одинаковые *value*. Например, в вычислениях *min()* и *max()* запись с наименьшим или наибольшим ключом будет возвращена, если найдутся дублированные (одинаковые) значения.

Например:

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

1.9. Поиск общих элементов в двух словарях

Задача

У вас два словаря, и вы хотите выяснить, что у них общего (одинаковые ключи, значения и т.п.)

Решение

Рассмотрим два словаря:

```
a = {
    'x' : 1,
    'y' : 2,
    'z' : 3
}

b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

Чтобы найти общие элементы, просто выполните обычный набор операций с использованием методов *keys()* и *items()*. Например:

```
# Находим общие ключи
a.keys() & b.keys() # { 'x', 'y' }
# Находим ключи, которые есть в a, но которых нет в b
a.keys() - b.keys() # { 'z' }
# Находим общие пары (key,value)
a.items() & b.items() # { ('y', 2) }
```

Операции такого типа также могут быть использованы для изменения или фильтрования содержимого словаря. Предположим, например, что вы хотите создать новый словарь, в котором некоторые ключи удалены. Взгляните на этот пример кода генератора словаря (dictionary comprehension):

```
# Создаём новый словарь, из которого удалены некоторые ключи
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c - это {'x': 1, 'y': 2}
```

Обсуждение

Словарь — это отображение множества ключей на множество значений. Метод словаря `keys()` возвращает объект просмотра ключей (`keys-view object`). Малоизвестная особенность этих объектов заключается в том, что они поддерживают набор операций над множествами (объединения, пересечения, разности и т.п.) Так что если вам нужно выполнить этот набор операций над ключами словаря, вы можете использовать объект просмотра напрямую, без предварительного конвертирования в множество.

Метод словаря `items()` возвращает объект просмотра элементов, состоящий из пар (`key, value`). Этот объект поддерживает похожий набор операций и может быть использован для выполнения таких операций, как поиск того, какие пары ключ-значение являются общими для двух словарей.

Хотя метод словаря `values()` похож на предыдущие, он не поддерживает операции над множествами, описанные выше в этом рецепте. Это происходит, в частности, по причине того, что, в отличие от ключей, элементы объекта просмотра значений могут и не быть уникальными. Один этот факт делает бесполезным применение к ним операций над множествами. Если же, однако, вы вынуждены выполнить такие операции, этого можно добиться простым путём предварительного конвертирования значений в множество.

1.10. Удаление дубликатов из последовательности с сохранением порядка элементов

Задача

Вы хотите исключить дублирующиеся значения из последовательности, но при этом сохранить порядок следования оставшихся элементов.

Решение

Если значения в последовательности являются хэшируемыми, задача может быть легко решена с использованием множества и генератора. Например:

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

Вот пример использования этой функции:

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

Это будет работать только в том случае, если элементы последовательности хэшируются. Если вы пытаетесь удалить дубликаты в последовательности из нехэшируемых типов (таких как словари), вы можете внести небольшое изменение в этот рецепт. Например, такое:

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

Аргумент *key* здесь нужен для определения функции, которая конвертирует элементы последовательности в хэшируемый тип, подходящий для поиска дубликатов. Вот как это работает:

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

Последнее решение также отлично работает, если вам нужно удалить дубликаты, базируясь на значении одного поля или атрибута или более крупной структуры данных.

Обсуждение

Если вы просто хотите удалить дубликаты, то часто достаточно будет создать множество. Например:

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

Однако этот поход не сохраняет какой бы то ни было порядок, поэтому результат будет перемешан. Показанные выше решения помогают избежать этого.

Использование функции-генератора в этом рецепте отражает тот факт, что вы наверняка хотите написать функцию максимально широкого назначения, а не напрямую привязанную к обработке списков. Например, если вы хотите читать файл, удаляя дублирующиеся строки, вы можете сделать так:

```
with open(somefile, 'r') as f:
    for line in dedupe(f):
        ...
```

Передача функции в аргументе *key* имитирует похожую возможность во встроенных функциях, таких как *sorted()*, *min()* и *max()*. См., например, рецепты [1.8](#) и [1.13](#).

1.11. Присваивание имён срезам

Задача

Ваша программа превратилась в нечитабельную массу индексов срезов, и вы хотите всё это расчистить.

Решение

Предположим, что у вас есть код, который вытаскивает определенные поля с данными из строковых записей с фиксированным набором полей (то есть из файла с плоской структурой или похожего формата):

```
##### 0123456789012345678901234567890123456789012345678901234567890 '
record = '.....100 .....513.25 ..'
cost = int(record[20:32]) * float(record[40:48])
```

Вместо этого вы вполне можете присвоить срезам имена:

```
SHARES = slice(20,32)
PRICE  = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

В последнем примере вы избежали появления кучи загадочных индексов, и код стал проще и яснее.

Обсуждение

Общее правило таково: написание кода с большим количеством неоформленных индексов ведет к проблемам с читабельностью и поддерживаемостью. Например, если вы вернетесь к такому коду через год, то наверняка не сразу вспомните, как и о чём вы думали, когда всё это писали. Приведённое выше решение — простой путь к более ясному обозначению того, что делает ваш код.

Встроенная функция `slice()` создает объект среза, который может быть использован везде, где можно было использовать обычные срезы. Например:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
```

```
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10, 11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

Если у вас есть экземпляр `slice`, сохранённый в переменной `s`, вы можете получить больше информации о нём, если посмотрите на атрибуты `s.start`, `s.stop` и `s.step`. Например:

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
>>> a.step
2
>>>
```

Также вы можете наложить срез на последовательность определенного размера, используя его метод `indices(size)`. Он возвращает кортеж (`start`, `stop`, `step`), где все значения соответственно ограничены, чтобы вписаться в границы (чтобы избежать возбуждения исключений `IndexError` при индексировании). Например:

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
d
>>>
```

1.12. Определение наиболее часто встречающихся элементов в

последовательности

Проблема

У вас есть последовательность элементов, и вы хотите узнать, какие элементы встречаются в ней чаще всего.

Решение

Класс `collections.Counter` разработан как раз решения для подобных задач. В нем даже есть удобный метод `most_common()`, который сразу выдаст вам ответ.

Чтобы проиллюстрировать это, предположим, что у вас есть список слов, и вы хотите найти наиболее часто встречающееся. Вот как можно это сделать:

```
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]

from collections import Counter

word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Выведем [('eyes', 8), ('the', 5), ('look', 4)]
```

Обсуждение

На входе объектам класса `Counter` можно скормить любую последовательность хэшируемых элементов. В основе `Counter` лежит словарь, который отображает элементы на количество вхождений.

Например:

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

Если вы хотите увеличить счёт вручную, используйте сложение:

```
>>> morewords = ['why', 'are', 'you', 'not', 'looking', 'in', 'my', 'eyes']
>>> for word in morewords:
...     word_counts[word] += 1
...
>>> word_counts['eyes']
9
>>>
```

Или же вы можете использовать метод *update()*:

```
>>> word_counts.update(morewords)
>>>
```

Малоизвестная возможность экземпляров *Counter* состоит в том, что они могут быть легко скомбинированы с использованием разнообразных математических операций. Например:

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
         'my': 1, 'why': 1})

>>> # Объединяем счётчики
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
         'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
         'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Вычитаем счётчики
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1})
>>>
```

Нет смысла упоминать, что объекты *Counter* — невероятно полезный инструмент для практически любых задач, где вам нужно перевести данные в

табличную форму и посчитать их. Рекомендуем использовать этот способ, а не писать вручную решения на основе словарей.

1.13. Сортировка списка словарей по общему ключу

Задача

У вас есть список словарей, и вы хотите отсортировать записи согласно одному или более значениям.

Решение

Сортировка структур этого типа легко выполняется с помощью функции *itemgetter* из модуля *operator*. Предположим, вы выполнили запрос к таблице базы данных, чтобы получить список зарегистрированных пользователей вашего сайта, и получили в ответ вот такую структуру данных:

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

Можно достаточно легко вывести эти строки с упорядочением по любому из полей, общих для всех словарей. Например:

```
from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

Вышеприведенный код выведет следующее:

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
```

```
{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

Функция *itemgetter()* также может принимать несколько ключей. Пример кода:

```
rows_by_lfname = sorted(rows, key=itemgetter('lname', 'fname'))
print(rows_by_lfname)
```

Вышеприведенный код выведет:

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

Обсуждение

В этом примере строки передаются встроенной функции *sorted()*, которая принимает именованный аргумент *key*. Этот аргумент должен быть вызываемым объектом, который принимает один элемент из *rows* и возвращает значение, которое будет использовано в качестве основы для сортировки. Функция *itemgetter()* создает такой вызываемый объект. (**Прим. пер.: вызываемый объект — это объект, который имеет метод `__call__`.**)

Функция *operator.itemgetter()* принимает в качестве аргументов индексы, которые используются для извлечения нужных значений из записей в *rows*. Это может быть ключ словаря, номер элемента в списке или любое другое значение, которое может быть скормлено методу *__getitem__()*. Если вы передадите несколько индексов функции *itemgetter()*, вызываемый объект, который она создаст, вернет кортеж со всеми элементами, и функция *sorted()* упорядочит выводимые элементы в соответствии с отсортированным порядком кортежей. Это может быть полезно, если вы хотите провести сортировку сразу по нескольким полям (в примере это имя и фамилия):

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'], r['fname']))
```

Это решение в большинстве случаев работает отлично. Однако решение с

использованием *itemgetter()* обычно выполняется быстрее. Так что обратите на него внимание, если производительность в фокусе вашего внимания.

Последнее по порядку, но не по значению: не забудьте, что описанная в этом рецепте техника может быть применена к таким функциям, как *min()* и *max()*. Например:

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14. Сортировка объектов, не поддерживающих сравнение

Задача

Вы хотите отсортировать объекты одного класса, но они не поддерживают операции сравнения.

Решение

Встроенная функция *sorted()* принимает аргумент *key*, в котором может быть передан вызываемый объект, который будет возвращать некоторое значение из объектов, которое *sorted()* будет использовать для сравнения этих объектов. Например, если у вас в приложении есть последовательность экземпляров класса *User*, и вы хотите отсортировать их по атрибуту *user_id*, то вы могли бы предоставить вызываемый объект, который принимает экземпляр класса *User* и возвращает атрибут *user_id*. Например:

```
>>> class User:
...     def __init__(self, user_id):
...         self.user_id = user_id
...     def __repr__(self):
...         return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
```

```
[User(3), User(23), User(99)]  
>>>
```

Вместо лямбды можно применить альтернативный подход с использованием `operator.attrgetter()`:

```
>>> from operator import attrgetter  
>>> sorted(users, key=attrgetter('user_id'))  
[User(3), User(23), User(99)]  
>>>
```

Обсуждение

Использовать или не использовать лямбду или `attrgetter()` — вопрос личных предпочтений. Однако `attrgetter()` часто оказывается немного быстрее, а также позволяет одновременно извлекать несколько полей. Это аналогично использованию `operator.itemgetter()` для словарей (см. [рецепт 1.13.](#)) Например, если экземпляры класса `User` также имеют атрибуты `first_name` и `last_name`, вы можете выполнить вот такую сортировку:

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

Также стоит отметить, что использованный в этом рецепте приём может быть применён к таким функциям, как `min()` и `max()`. Например:

```
>>> min(users, key=attrgetter('user_id'))  
User(3)  
>>> max(users, key=attrgetter('user_id'))  
User(99)  
>>>
```

1.15. Группирование записей на основе полей

Задача

У вас есть последовательность словарей или экземпляров, и вы хотите итерировать по данным, сгруппированным по значению конкретного поля (например, на дате).

Решение

Функция `itertools.groupby()` особенно полезна для такого типа группирования данных. Предположим, что у вас есть список словарей:

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

Предположим также, что вы хотите проитерировать по группам данных, объединенных общей датой. Проведем сортировку по нужному полю (в данном случае по дате), а потом применим `itertools.groupby()`:

```
from operator import itemgetter
from itertools import groupby

# Сначала сортируем по нужным полям
rows.sort(key=itemgetter('date'))

# Итерируем в группах
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

Вывод будет таким:

```
07/01/2012
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
{'date': '07/02/2012', 'address': '5800 E 58TH'}
{'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
{'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
{'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
{'date': '07/04/2012', 'address': '5148 N CLARK'}
{'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

Обсуждение

Функция `groupby()` работает так: сканирует последовательность и ищет последовательные «партии» одинаковых значений (или значений, возвращенных переданной через `key` функцией). В каждой итерации функция возвращает значение вместе с итератором, который выводит все элементы в группу с одинаковым значением.

Важным предварительным шагом тут является сортировка данных по интересующему нас полю. Поскольку `groupby()` проверяет только последовательные элементы, без предварительной сортировки группировка записей выполнена не будет.

Если ваша цель — просто сгруппировать данные вместе в крупную структуру данных с произвольным доступом, то вам больше поможет функция `defaultdict()`, которая создает «мультисловарь», как было описано в [рецепте 1.6](#). Например:

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

Это позволяет легко получить доступ к записям для каждой даты:

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

В последнем примере предварительная сортировка записей не обязательна. Но если вы не заботитесь о потреблении памяти, то может оказаться быстрее сделать это с помощью предварительной сортировки и итерирования с использованием `groupby()`.

1.16. Фильтрование элементов последовательности

Задача

У вас есть данные внутри последовательности, и вы хотите извлечь значения или сократить последовательность по какому-либо критерию.

Решение

Самый простой способ фильтрования последовательности — использовать генератор списка (list comprehension). Например:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

Потенциальная проблема с использованием генераторов списков заключается в том, что они могут создать большой результат, если размер входных данных тоже большой. Если это вас беспокоит, вы можете использовать выражения-генераторы для итеративного возврата отфильтрованных значений. Например:

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
...     print(x)
...
1
4
10
2
3
>>>
```

Иногда критерий фильтрования не может быть легко выражен в форме генератора списка или выражения-генератора. Предположим, например, что процесс фильтрования включает обработку исключений или какой-то другой сложный момент. Чтобы справиться с этим, поместите фильтрующий код в функцию и используйте встроенную функцию *filter()*. Например:

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']
```

```
def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False

ivals = list(filter(is_int, values))
print(ivals)
# Выведем [ '1', '2', '-3', '4', '5' ]
```

filter() создает итератор, так что если вы хотите получить список результатов, не забудьте использовать *list()*, как показано выше.

Обсуждение

Генераторы списков и выражения-генераторы часто являются самым лёгким и прямым способом фильтрования простых данных. Но у них также есть дополнительная способность — одновременно изменять данные. Например:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

Одна из разновидностей фильтрования включает замену значений, которые не подходят под определенный критерий, другими значениями (вместо отбраковки неподходящих). Например, вместо простого поиска положительных значений, вы также хотите обрезать «плохие» значения, чтобы они попадали в определенный диапазон. В большинстве случаев это легко сделать с помощью перемещения критерия фильтрования в условное выражение:

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

Другой важный инструмент для фильтрации — *itertools.compress()*, который

принимает итерируемый объект вместе с последовательностью-селектором из булевых значений. На выходе функция выдает все элементы итерируемого объекта, для которых совпадающий элемент в селекторе — True. Это может быть полезно, если вы пытаетесь применить результаты фильтрования одной последовательности к другой связанной последовательности. Например, у вас есть две колонки данных:

```
addresses = [  
    '5412 N CLARK',  
    '5148 N CLARK',  
    '5800 E 58TH',  
    '2122 N CLARK'  
    '5645 N RAVENSWOOD',  
    '1060 W ADDISON',  
    '4801 N BROADWAY',  
    '1039 W GRANVILLE',  
]  
  
counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

Теперь предположим, что вы хотите создать список всех адресов, где соответствующие значение из `counts` больше 5. Вот как это можно сделать:

```
>>> from itertools import compress  
>>> more5 = [n > 5 for n in counts]  
>>> more5  
[False, False, True, False, False, True, True, False]  
>>> list(compress(addresses, more5))  
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']  
>>>
```

Ключевой момент — сначала создать последовательность булевых значений, которые будут указывать, какие элементы удовлетворяют заданному условию. Далее функция `compress()` выберет элементы, соответствующие значениям `True`.

Как и `filter()`, функция `compress()` возвращает итератор. Поэтому если вы хотите на выходе получить список, вам придется использовать `list()`.

1.17. Извлечение подмножества из словаря

Задача

Вы хотите создать словарь, который будет подмножеством другого словаря.

Решение

Эту задачу можно легко решить с помощью генератора словаря (dictionary comprehension). Например:

```
prices = {  
    'ACME': 45.23,  
    'AAPL': 612.78,  
    'IBM': 205.55,  
    'HPQ': 37.20,  
    'FB': 10.75  
}  
  
# Создать словарь всех акций с ценами больше 200  
p1 = { key:value for key, value in prices.items() if value > 200 }  
  
# Создать словарь акций технологических компаний  
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }  
p2 = { key:value for key,value in prices.items() if key in tech_names }
```

Обсуждение

Большую часть того, что можно сделать с помощью генераторов словарей, можно осуществить путём создания последовательности кортежей и передачи их в функцию `dict()`. Например:

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

Однако решение на основе генератора словаря яснее и работает немного быстрее (в рассмотренном выше примере генератор отработал в два раза быстрее).

Иногда существует множество путей решить задачу. Например, второй пример можно переписать так:

```
# Создать словарь акций технологических компаний  
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }  
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

Однако подсчёт времени выполнения открывает нам, что это решение почти в 1,6 раза медленнее, чем первое. Если производительность для вас важна, обычно стоит потратить немного времени на изучение таких вопросов. См. рецепт [14.13.](#), чтобы получить детальную информацию о подсчёте времени и профилировании.

1.18. Отображение имен на последовательность элементов

Задача

У вас есть код, который осуществляет доступ к элементам в списке или кортеже по позиции. Однако такой подход часто делает программу нечитабельной. Также вы можете захотеть уменьшить зависимость от позиции в структуре данных путём перехода к принципу доступа к элементам по имени.

Решение

`collections.namedtuple()` предоставляет такую возможность, добавляя лишь минимальный оверхед по сравнению с использованием обычного кортежа. `collections.namedtuple()` – это фабричный метод, который возвращает подкласс стандартного типа Python `tuple` (кортеж). Вы скармливаете этому методу имя типа и поля, которые он должен иметь. Он возвращает класс, который может порождать экземпляры с полями, которые вы определили, а также значениями этих полей, которые вы передадите при порождении.

Например:

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

Хотя экземпляр `namedtuple` (именованного кортежа) выглядит так же, как и

обычный экземпляр класса, он взаимозаменяется с кортежем и поддерживает все обычные операции кортежей, такие как индексирование и распаковка.

Например:

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

Самый частый случай использования именованного кортежа — отвязка вашего кода от работы с позициями элементов, которыми он манипулирует. Скажем, если вы получаете большой список кортежей в ответ на запрос к базе данных, а потом манипулируете ими через позиционное обращение к элементам, ваш код может сломаться, если вы, например, добавите новую колонку в таблицу. Этого можно избежать, если вы сначала превратите полученные кортежи в именованные кортежи.

Чтобы проиллюстрировать это, приведём пример кода, использующего обычные кортежи:

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

Использование позиционного обращения к элементам часто делает код немного менее выразительным и более зависимым от структуры записей. А вот версия с использованием именованного кортежа:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

Естественно, вы можете избежать явной конвертации в именованный кортеж `Stock`, если последовательность `records` из примера уже содержит такие экземпляры.

Обсуждение

Возможное использование именованного кортежа — замена словаря, который требует больше места для хранения. Так что если создаете крупные структуры данных с использованием словарей, применение именованных кортежей будет более эффективным. Однако не забудьте, что именованные кортежи неизменяемы (в отличие от словарей). Например:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Если вам нужно изменить любой из атрибутов, это может быть сделано с помощью метода `_replace()`, которым обладают экземпляры именованных кортежей. Он создает полностью новый именованный кортеж, в котором указанные значения заменены. Например:

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

Тонкость использования метода `_replace()` заключается в том, что он может стать удобным способом наполнить значениями именованный кортеж, у которого есть опциональные или отсутствующие поля. Чтобы сделать это, создайте прототип кортежа, содержащий значения по умолчанию, а затем применяйте `_replace()` для создания новых экземпляров с замененными значениями. Например:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Создание экземпляра прототипа
```

```
stock_prototype = Stock('', 0, 0.0, None, None)

# Функция для преобразования словаря в Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

Вот пример работы этого кода:

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

Последнее, но важное замечание: стоит отметить, что если вашей целью является создание эффективной структуры данных, которая позволяет менять различные атрибуты экземпляров, использование именованных кортежей — не лучший вариант. Вместо них стоит определить класс с использованием `__slots__` (см. [рецепт 8.4.](#))

1.19. Одновременное преобразование и сокращение (свёртка) данных

Задача

Вам нужно выполнить функцию сокращения (т.е. `sum()`, `min()`, `max()`), но сначала необходимо преобразовать или отфильтровать данные.

Решение

Есть весьма элегантное решение для объединения сокращения (свёртки) и преобразования данных — выражение-генератор в аргументе. Например, если вы хотите подсчитать сумму квадратов, попробуйте следующее:

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

Вот еще несколько примеров:

```

# Определяем, есть ли файлы .py в каталоге
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')

# Выводит кортеж как CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))

# Сокращение (reduction) данных по полям в структуре данных
portfolio = [
    {'name': 'GOOG', 'shares': 50},
    {'name': 'YHOO', 'shares': 75},
    {'name': 'AOL', 'shares': 20},
    {'name': 'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)

```

Обсуждение

Решение демонстрирует тонкий синтаксический аспект выражений-генераторов, связанный с передачей их как единственного аргумента в функцию: повторяющиеся скобки не нужны. Например, следующие инструкции эквивалентны:

```

s = sum((x * x for x in nums))      # Передаем выражение-генератор
                                         # в качестве аргумента
s = sum(x * x for x in nums)        # Более элегантный синтаксис

```

Использование аргумента-генератора часто будет более эффективным и элегантным, нежели предварительное создание временного списка. Например, если вы не используете выражение-генератор, вы можете склониться к такой реализации:

```

nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])

```

Это работает, но вводит лишний шаг и создает лишний список. Для небольшого списка из примера это не имеет значения, но если *nums* был огромным, вы получите крупную временную структуру данных, которая будет использована только один раз, а потом выброшена. Решение с генератором

обрабатывает данные итеративно и потому намного более эффективно с точки зрения использования памяти.

Некоторые функции сокращения (свёртки), такие как `min()` и `max()`, принимают аргумент `key`, что может оказаться полезным в ситуациях, когда вы склоняетесь к использованию генератора. Например, в этом примере вы можете попробовать альтернативный подход:

```
# Изначальный: возвращает 20
min_shares = min(s['shares'] for s in portfolio)

# Альтернативный: возвращает {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20. Объединение нескольких отображений в одно

Задача

У вас есть много словарей или отображений, которые вы хотите логически объединить в одно отображение, чтобы выполнить некоторые операции, такие как поиск значений или проверка существования ключей.

Решение

Предположим, у вас есть два словаря:

```
a = {'x': 1, 'z': 3}
b = {'y': 2, 'z': 4}
```

А теперь предположим, что вы хотите провести поиски, в ходе которых вам нужно проверить оба словаря (то есть сначала проверить в словаре `a`, а потом в `b`, если в первом словаре искомое не найдено). Простой способ сделать это — использовать класс `ChainMap` из модуля `collections`. Например:

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x']) # Выводит 1 (из a)
print(c['y']) # Выводит 2 (из b)
print(c['z']) # Выводит 3 (из a)
```

Обсуждение

ChainMap принимает несколько отображений и делает их логически единым целым. Однако в буквальном смысле они не сливаются. Вместо этого *ChainMap* просто содержит список отображений и переопределяет обычные операции над словарями для сканирования этого списка. Большинство операций работают. Например:

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

В случае появления одинаковых ключей будут использованы значения из первого словаря. Например, *c['z']* в примере всегда будет ссылаться на значение из словаря *a*, а не из *b*.

Операции, которые изменяют отображение, всегда действуют на первое отображение в списке. Например:

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

ChainMap особенно полезны для работы со значениями, принадлежащими областям видимости, такими как переменные языка программирования (глобальные, локальные и т.п.) На самом деле даже существуют методы, которые всё упрощают:

```
>>> values = ChainMap()
>>> values['x'] = 1
>>> # Добавляем новое отображение
>>> values = values.new_child()
>>> values['x'] = 2
```

```
>>> # Добавляем новое отображение
>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Удаляем последнее отображение
>>> values = values.parents
>>> values['x']
2
>>> # Удаляем последнее отображение
>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>
```

В качестве альтернативы *ChainMap* вы можете обдумать слияние словарей с использованием метода `update()`. Например:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

Это работает, но требует от вас создания полностью нового объекта словаря (или необратимого изменения одного из существующих). В этом случае при изменении одного из первоначальных словарей изменения не затронут новый объект объединенного словаря. Например:

```
>>> a['x'] = 13
>>> merged['x']
1
```

ChainMap использует первоначальные словари, поэтому не подвержен такому поведению. Например:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x'] # Изменение происходит и в объединённых словарях
42
>>>
```

2. Строки и текст

Практически любая полезная программа включает тот или иной вид обработки текста: от парсинга данных до генерации вывода. Эта глава рассматривает обычные задачи манипулирования текстом, такие как разбивка строк, поиск, подстановка, лексический анализ и парсинг. Многие из этих задач могут быть легко решены с использованием встроенных строковых методов. Однако более сложные операции могут потребовать использования регулярных выражений или создания полноценного парсера. Все эти темы разобраны в данной главе. Также мы обратим внимание на несколько тонких аспектов работы с Unicode.

2.1. Разрезание строк, разделенных различными разделителями

Задача

Вам нужно разделить строку на поля, но разделители (и пробелы вокруг них) внутри строки разные.

Решение

Функция `re.split()` будет в этом случае весьма полезной, поскольку вы сможете определить многочисленные шаблоны разделителей. Например, как показано в решении, разделитель может быть либо запятой (,), точкой с запятой (;) или пробелом, за которым следует любое количество дополнительных пробелов. Какой бы из этих шаблонов ни был найден, совпадение становится разделителем. Результатом будет просто список

полей — точно такой же, какой создает строковый метод `str.split()`.

При применении `re.split()` вы должны быть осторожными, если шаблон регулярного выражения использует группу, заключенную в скобки. При использовании групп совпадший с шаблоном текст также включается в результат. Например:

```
>>> fields = re.split(r'(;| |\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',', 'foo']
>>>
```



Получение символов-разделителей может быть полезным в некоторых обстоятельствах. Например, вам могут потребоваться эти символы позже — для переформатирования выводимой строки:

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', ',', ''']

>>> # Переформатируем строку, используя те же разделители
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

Если вы не хотите, чтобы разделители попали в результат, но при этом вам нужно применить группы в шаблоне регулярного выражения, убедитесь, что вы используете незахватывающую (noncapture) группу, которая определяется так: `(?:...)`. Например:

```
>>> re.split(r'(?:| ;|\s)\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

2.2. Поиск текста в начале и в конце строки

Задача

Вам нужно проверить начало или конец строки на присутствие неких текстовых шаблонов, таких как расширения файлов, схемы URL и т.д.

Решение

Простой способ проверить начало или конец строки — применить методы `str.startswith()` или `str.endswith()`. Например:

```
>>> filename = 'spam.txt'  
>>> filename.endswith('.txt')  
True  
>>> filename.startswith('file:')  
False  
>>> url = 'http://www.python.org'  
>>> url.startswith('http: ')  
True  
>>>
```

Если вам нужно проверить несколько вариантов, передайте кортеж с ними в `startswith()` или `endswith()`:

```
>>> import os  
>>> filenames = os.listdir('.')  
>>> filenames  
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]  
>>> [name for name in filenames if name.endswith(('.c', '.h'))]  
['foo.c', 'spam.c', 'spam.h']  
>>> any(name.endswith('.py') for name in filenames)  
True  
>>>
```

А вот другой пример:

```
from urllib.request import urlopen  
  
def read_data(name):  
    if name.startswith(('http:', 'https:', 'ftp:')):  
        return urlopen(name).read()  
    else:  
        with open(name) as f:  
            return f.read()
```

Любопытно, что в этом случае на вход нужно подавать именно кортеж. Если так случилось, что варианты выбора собраны у вас в списке или множестве, сначала сконвертируйте их с помощью `tuple()`. Например:

```
>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, not list
>>> url.startswith(tuple(choices))
True
>>>
```

Обсуждение

Методы `startswith()` и `endswith()` предоставляют весьма удобный способ проверки префиксов и окончаний. Такие же операции можно осуществить с помощью срезов, но это намного менее элегантно. Например:

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or url[:6] == 'https:' or url[:4] == 'ftp:'
True
>>>
```

Вы также можете склониться к использованию регулярных выражений в качестве альтернативы. Например:

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
<_sre.SRE_Match object at 0x101253098>
>>>
```

Такой подход работает, но часто это будет огнем из пушки по воробьям. Использование вышеописанного рецепта проще и работает быстрее.

И последнее: методы `startswith()` и `endswith()` отлично работают вместе с другими операциями, такими как обычные методы сокращения (свёртки) данных. Например, это выражение проверяет каталог на присутствие файлов определенных типов:

```
if any(name.endswith('.c', '.h')) for name in listdir(dirname)):
```

2.3. Поиск строк с использованием масок оболочки (shell)

Задача

Вы хотите найти текст, используя те же маски, которые обычно используются в оболочках Unix (например, `*.py`, `Dat[0-9]*.csv` и т.д.)

Решение

Модуль `fnmatch` предоставляет две функции: `fnmatch()` и `fnmatchcase()`, которые можно использовать для такого поиска. Всё просто:

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

По умолчанию `fnmatch()` использует те же чувствительные к регистру правила, как и файловая система текущей операционной системы (то есть правила меняются от системы к системе). Например:

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False

>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

Если это различие важно, используйте метод `fnmatchcase()`. Он ищет именно такие совпадения заглавных и строчных букв, которые вы предоставите:

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
```

```
>>>
```

Часто упускается из вида возможность использования этих функций на строках, получаемых при обработке данных, или на строках, не являющихся именами файлов. Например, у вас есть список адресов:

```
addresses = [  
    '5412 N CLARK ST',  
    '1060 W ADDISON ST',  
    '1039 W GRANVILLE AVE',  
    '2122 N CLARK ST',  
    '4802 N BROADWAY',  
]
```

Вы можете написать такой генератор списка (list comprehension):

```
>>> from fnmatch import fnmatchcase  
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]  
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']  
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] *CLAR')]  
['5412 N CLARK ST']  
>>>
```

Обсуждение

Поиск совпадений с использованием *fnmatch* занимает нишу между возможностями простых строковых методов и полной мощью регулярных выражений. Если вам нужен простой механизм для применения масок в обработке данных, он часто будет подходящим решением.

Если же вы пишете код для поиска имён файлов, используйте модуль *glob* (см. рецепт 5.13.)

2.4. Поиск совпадений и поиск текстовых паттернов

Задача

Вы хотите отыскать совпадение или провести поиск по определенному шаблону.

Решение

Если текст, который вы хотите найти, является простым литералом, в большинстве случаев вам подойдут базовые строковые методы, такие как `str.find()`, `str.endswith()`, `str.startswith()` и другие подобные. Например:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> # Точное совпадение
>>> text == 'yeah'
False

>>> # Совпадение по началу или концу
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False

>>> # Поиск места первого вхождения
>>> text.find('no')
10
>>>
```

Для более сложного поиска совпадений используйте регулярные выражения и модуль `re`. Чтобы проиллюстрировать базовые механизмы использования регулярных выражений, предположим, что вы хотите найти даты, определенные цифрами, такие как «11/27/2012». Вот пример того, как вы можете это сделать:

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
>>> # Простое сопоставление: \d+ означает совпадение одной или более цифр
>>> if re.match(r'\d+/\d+/\d+', text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
...     print('yes')
... else:
...     print('no')
...
no
>>>
```

Если вы собираетесь много раз искать по одному и тому же шаблону, часто окупается предварительная компиляция шаблона регулярного выражения в объект шаблона. Например:

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if datepat.match(text2):
...     print('yes')
... else:
...     print('no')
...
no
>>>
```

match() всегда пытается найти совпадения в начале строки. Если вы хотите провести поиск по всем случаям соответствия шаблону, используйте метод *findall()*. Например:

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

При составлении регулярных выражений часто нужно использовать захватывающие группы, заключая части шаблона в скобки. Например:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
```

Захватывающие группы часто упрощают последующую обработку найденного текста, поскольку содержимое каждой группы может быть извлечено отдельно. Например:

```
>>> m = datepat.match('11/27/2012')
>>> m
<sre.SRE_Match object at 0x1005d2750>

>>> # Извлекаем содержимое каждой группы
>>> m.group(0)
```

```
'11/27/2012'  
>>> m.group(1)  
'11'  
>>> m.group(2)  
'27'  
>>> m.group(3)  
'2012'  
>>> m.groups()  
('11', '27', '2012')  
>>> month, day, year = m.groups()  
>>>  
  
>>> # Найти все совпадения ( обратите внимание на  
>>> # разрезание на кортежи )  
>>> text  
'Today is 11/27/2012. PyCon starts 3/13/2013.'  
>>> datepat.findall(text)  
[('11', '27', '2012'), ('3', '13', '2013')]  
>>> for month, day, year in datepat.findall(text):  
...     print('{}-{}-{}'.format(year, month, day))  
...  
2012-11-27  
2013-3-13  
>>>
```

Метод `findall()` проходит по тексту и находит все совпадения, возвращая их в списке. Если вы хотите искать совпадения итеративно, используйте метод `finditer()`:

```
>>> for m in datepat.finditer(text):  
...     print(m.groups())  
...  
('11', '27', '2012')  
('3', '13', '2013')  
>>>
```

Обсуждение

Вводного курса в теорию регулярных выражений в этой книге вы не найдёте. Однако этот рецепт демонстрирует простейшие примеры использования модуля `re` для поиска совпадений в тексте. Самые основные приёмы — компилирование шаблонов с использованием `re.compile()` и последующее использование таких методов, как `match()`, `findall()` или `finditer()`.

При составлении шаблонов часто нужно использовать «сырые» (raw) строки, такие как `r'(\d+)/(\d+)/(\d+)'`. Такие строки оставляют символы обратных

слэшей необработанными, что может быть полезно в контексте применения регулярных выражений. С другой стороны, вы можете использовать двойные обратные слэши: '`\d+/\d+/\d+`'.

Учтите, что метод `match()` проверяет только начало строки. Возможно, что он найдет вещи, которые вы не ожидаете. Например:

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>
```

Если вам нужно точное совпадение, убедитесь, что шаблон включает символ завершения (\$), как в примере ниже:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)\$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

И последнее: если вы просто проводите простые операции поиска, вы часто можете пропустить шаг компиляции и использовать функции уровня модуля из модуля `re`. Например:

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

Обратите внимание, что если вы проводите много операций поиска совпадений, часто окупается компилирование шаблона и многократное его использование. Функции уровня модуля поддерживают кэш недавно скомпилированных шаблонов, так что вы не получите огромного выигрыша в производительности, но вы сэкономите несколько обращений и избежите лишней обработки, используя ваш собственный скомпилированный шаблон.

2.5. Поиск и замена текста

Задача

Вы хотите найти в строке и заменить текст, соответствующий некому шаблону.

Решение

Для простых литеральных шаблонов используйте метод `str.replace()`.

Например:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'  
  
>>> text.replace('yeah', 'yep')  
'yep, but no, but yep, but no, but yep'  
>>>
```

Для более сложных шаблонов используйте функции/методы `sub()` из модуля `re`. Предположим, вы хотите перезаписать даты, чтобы перевести из их формата “11/27/2012” в “2012-11-27.” Вот пример того, как это можно сделать:

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'  
>>> import re  
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)  
'Today is 2012-11-27. PyCon starts 2013-3-13.'  
>>>
```

Первый аргумент, передаваемый в `sub()`, это шаблон для поиска, а второй — шаблон, которым нужно заменять найденное. Цифры, перед которыми поставлен обратный слэш (такие как `\3`), ссылаются на номера захватывающих групп в шаблоне.

Если вы собираетесь многократно выполнять подстановку по одному и тому же шаблону, рекомендуем скомпилировать его для увеличения производительности. Например:

```
>>> import re  
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')  
>>> datepat.sub(r'\3-\1-\2', text)  
'Today is 2012-11-27. PyCon starts 2013-3-13.'  
>>>
```

В случае более сложных подстановок можно определить подстановочную функцию обратного вызова (коллбэк, callback). Например:

```
>>> from calendar import month_abbr
>>> def change_date(m):
...     mon_name = month_abbr[int(m.group(1))]
...     return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

На вход подстановочному коллбэку в аргументе передается объект поиска совпадений, возвращенный функцией *match()* или *find()*. Используйте метод *.group()* для извлечения определенных частей совпадения. Функция должна возвращать изменённый текст.

Если вы хотите знать, сколько подстановок было сделано (в дополнение к получению изменённого текста), используйте *re.subn()*. Например:

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

Обсуждение

В поиске совпадений с помощью регулярных выражений не особо много чего-то ещё, помимо показанного примера с использованием метода *sub()*. Самое сложное — это составление шаблонов регулярных выражений, и это мы оставляем читателю в качестве упражнений.

2.6. Поиск и замена текста без учета регистра

Задача

Вам необходимо найти и, возможно, заменить текст, не обращая внимания на регистр букв.

Решение

Для выполнения действий над текстом без учёта регистра вам понадобится модуль `re` и флаг `re.IGNORECASE`, который можно применять в различных операциях. Например:

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'  
>>> re.findall('python', text, flags=re.IGNORECASE)  
['PYTHON', 'python', 'Python']  
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)  
'UPPER snake, lower snake, Mixed snake'  
>>>
```

Последний пример демонстрирует ограничение способа: текст замены не будет совпадать по регистру с заменяемым текстом. Если вам нужно исправить такое поведение, используйте поддерживающую функцию:

```
def matchcase(word):  
    def replace(m):  
        text = m.group()  
        if text.isupper():  
            return word.upper()  
        elif text.islower():  
            return word.lower()  
        elif text[0].isupper():  
            return word.capitalize()  
        else:  
            return word  
    return replace
```

А вот пример использования этой функции:

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECASE)  
'UPPER SNAKE, lower snake, Mixed Snake'  
>>>
```

Обсуждение

В простых случаях простого использования `re.IGNORECASE` достаточно для поиска совпадений без учёта регистра. Однако обратите внимание, что этого может оказаться недостаточно для некоторых случаев работы с Unicode, использующих выравнивание регистров (case folding). См. рецепт 2.10.

2.7. Определение регулярных выражений

для поиска кратчайшего совпадения

Задача

Вы пытаетесь найти совпадение по текстовому шаблону, используя регулярное выражение, но оно находит самое длинное из всех возможных совпадений. Вы же хотите найти самое короткое из всех возможных.

Решение

Эта проблема часто возникает при использовании шаблонов, которые пытаются найти текст, заключенный в пару открывающих и закрывающих разделителей (например, строку в кавычках). Рассмотрим следующий пример:

```
>>> str_pat = re.compile(r'\"(.*)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no.' Phone says "yes."]
>>>
```

В этом примере шаблон `r'\"(.*)\"'` пытается найти текст, заключенный в кавычки. Однако оператор `*` в регулярном выражении является жадным, поэтому поиск получается поиском самого длинного из возможных совпадений. Поэтому во втором примере с переменной `text2` неверно выполняется сопоставление двух строк в кавычках.

Чтобы исправить это, добавьте модификатор `?` после оператора `*` в шаблоне:

```
>>> str_pat = re.compile(r'\"(..*?)\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

Это делает поиск совпадений нежадным и выводит кратчайшее из найденных совпадений.

Обсуждение

Этот рецепт решает одну из часто встречающихся при написании регулярных выражений с символом точки (.) задач. В шаблоне точка соответствует любому символу за исключением символа новой строки. Однако если вы окружите точку открывающим и закрывающим текстом (таким как кавычки), поиск будет пытаться найти самое длинное из возможных совпадений. Это вызывает многочисленные случаи пропуска открывающего и закрывающего текста и включения в результаты самого длинного совпадения. Добавление ? сразу после таких операторов как * или + заставляет алгоритм поиска искать самое короткое совпадение.

2.8. Написание регулярного выражения для многострочных шаблонов

Задача

Вы пытаетесь провести поиск по блоку текстов с использованием регулярного выражения, но вам нужно, чтобы совпадение охватывало несколько строк.

Решение

Эта проблема обычно возникает в шаблонах, которые используют точку(.) для поиска совпадения с любым символом. Многие забывают, что точка не может совпадать с символом новой строки. Например, вы пытаетесь найти совпадения в комментариях в стиле языка С:

```
>>> comment = re.compile(r'/*(.*?)*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
... multiline comment */
...
...
>>>
>>> comment.findall(text1)
[ ' this is a comment ' ]
>>> comment.findall(text2)
[ ]
```

Чтобы исправить проблему, вам нужно добавить поддержку символов новой строки. Например:

```
>>> comment = re.compile(r'/*((?:.|\\n)*?)*/')
>>> comment.findall(text2)
[ ' this is a\\n multiline comment ']
>>>
```

В этом шаблоне `(?:.|\\n)` определяет незахватывающую группу (то есть выражение определяет группу для целей поиска совпадений, но эта группа не захватывается и не подсчитывается).

Обсуждение

Функция `re.compile()` принимает полезный в данном случае флаг `re.DOTALL`. Он заставляет `.` в регулярном выражении совпадать с любыми символами, включая символ новой строки. Например:

```
>>> comment = re.compile(r'/*(.*)*/', re.DOTALL)
>>> comment.findall(text2)
[ ' this is a\\n multiline comment ']
```

Использование флага `re.DOTALL` отлично работает в простых случаях, но это может давать сбои при работе с очень сложными шаблонами или сочетанием отдельных регулярных выражений, которые должны объединяться друг с другом для токенизации (как описано в [рецепте 2.18.](#)) Если у вас есть выбор, обычно лучше определить шаблон регулярного выражения так, чтобы он работал правильно без необходимости в дополнительных флагах.

2.9. Приведение текста в Unicode к стандартному представлению (нормализация)

Задача

Вы работаете со строками Unicode и хотите убедиться, что все эти строки имеют одинаковое внутреннее представление.

Решение

В Unicode некоторые символы могут быть представлены несколькими

допустимыми кодирующими последовательностями. Рассмотрим пример:

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalape\u00f1o'
>>> s2
'Spicy Jalape\u00f1o'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

Здесь текст “Spicy Jalapeño” представлен в двух формах. Первая использует полноценный символ “ñ” (U+00F1). Второй использует латинскую букву “n”, за которой следует дополняющий символ “~” (U+0303).

Такие множественные представления становятся проблемой для программ, которые занимаются сравнением строк. Чтобы это исправить, вы должны сначала нормализовать текст, то есть привести его к стандартному представлению с помощью модуля *unicodedata*:

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'

>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

Первый аргумент, передаваемый в *normalize()*, определяет режим нормализации текста. *NFC* означает, что символы должны быть полноценными (то есть по возможности использовать только одну кодирующую последовательность). *NFD* означает, что символы должны быть декомпозированными, то есть разделенными на комбинирующиеся символы.

Python также поддерживает режимы нормализации *NFKC* и *NFKD*, которые добавляют возможности совместимости, которые позволяют работать с определенными типами символов. Например:

```
>>> s = '\ufb01' # Один символ
>>> s
'fi'
>>> unicodedata.normalize('NFD', s)
'fi'

# Обратите внимание на разбивку объединённых букв
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

Обсуждение

Нормализация — это важная часть любой программы, в которой присутствует необходимость обработки текста в Unicode разумным и единообразным способом. Это особенно важно, когда обрабатываемые строки поступают из пользовательского ввода, кодировку которого вы практически никак не контролируете.

Нормализация (приведение) также может быть важной частью чистки и фильтрации текста. Предположим, например, что вы хотите удалить из текста диакритические знаки (возможно, для цели поиска совпадений):

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

Последний пример демонстрирует еще один важный аспект модуля *unicodedata*, а именно полезные функции для проверки принадлежности символов к определенным классам символов. Функция *combining()* проверяет, является ли символ объединяющимся. В этом модуле есть и другие функции для поиска символов определенных категорий, проверки цифр и так далее.

Unicode — весьма обширная тема. Для более подробной информации о нормализации посетите [соответствующую страницу](#) на сайте Unicode. Нед

Батчелдер также разместил на своем сайте [отличную презентацию](#) о решении проблем, связанных с Unicode в Python.

2.10. Использование символов Unicode в регулярных выражениях

Задача

Вы используете регулярные выражения для обработки текста, однако беспокоитесь о правильном взаимодействии с символами Unicode.

Решение

По умолчанию модуль `re` уже имеет некоторые зачаточные представления о некоторых типах символов Unicode. Например, `\d` совпадает с любым цифровым символом Unicode:

```
>>> import re
>>> num = re.compile('\d+')
>>> # ASCII-цифры
>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>

>>> # Арабские цифры
>>> num.match('\u0661\u0662\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

Если вам нужно включить специфические символы Unicode в шаблоны, вы можете использовать обычные последовательности для экранирования символов Unicode (например, `\uFFFF` или `\UFFFFFFF`). Например, вот регексп, который найдет совпадения со всеми символами на нескольких разных арабских страницах:

```
>>> arabic = re.compile('[\u0600-\u06ff\u0750-\u077f\u08a0-\u08ff]+')
```

При выполнении поиска совпадений следует нормализовывать и, по возможности, чистить текст, приводя его к стандартной форме (см. [рецепт 2.9.](#)) Так же нужно знать о некоторых специальных случаях. Например,

рассмотрим поведение нечувствительного к регистру поиска совпадений при объединении с приведением к одному регистру:

```
>>> pat = re.compile('stra\u00dfe', re.IGNORECASE)
>>> s = 'straße'
>>> pat.match(s) # Matches
<sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper()) # Doesn't match
>>> s.upper() # Case folds
'STRASSE'
>>>
```

Обсуждение

Смешивание Unicode и регулярных выражений — отличный способ взорвать себе голову. Если вы собираетесь серьезно в это погрузиться, установите не включенную в стандартную поставку Python библиотеку [reex](#), в которой есть полная поддержка приведения текстов в Unicode к одному регистру, а также множество других интересных возможностей, включая аппроксимирующй поиск совпадений.

2.11. Убиране нежелательных символов из строк

Задача

Вы хотите убрать ненужные символы, такие как пробелы в начале, конце или середине текстовой строки.

Решение

Метод `strip()` можно использовать для срезания символов в начале или конце строки. `lstrip()` и `rstrip()` выполняют срезание слева и справа соответственно. По умолчанию они срезают пробел, однако им можно передать и другие символы. Например:

```
>>> # Обрезание пробелов
>>> s = ' hello world \n'
>>> s.strip()
'hello world'
```

```
>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
' hello world'

>>>
>>> # Обрезание символов
>>> t = '-----hello-----'
>>> t.lstrip('-')
'hello-----'
>>> t.strip('-=')
'hello'
>>>
```

Обсуждение

Различные методы `strip()` часто используются при чтении и чистке данных для последующей обработки. Например, вы можете использовать их, чтобы избавиться от пробелов, удалить кавычки и т.д.

Обратите внимание, что срезание символов нельзя применить к тексту в середине строки. Например:

```
>>> s = ' hello world \n'
>>> s = s.strip()
>>> s
'hello world'
>>>
```

Если вам нужно что-то сделать с внутренним пробелом, вам нужно применить другой приём, такой как использование метода `replace()` или подстановку с использованием регулярного выражения. Например:

```
>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

Часто вам нужно сочетать срезание символов с другими видами итерационной обработки, таким как чтением строк данных из файла. Если это так, то стоит применить выражение-генератор:

```
with open(filename) as f:  
    lines = (line.strip() for line in f)  
    for line in lines:  
        ...
```

Здесь выражение `lines = (line.strip() for line in f)` работает как преобразователь данных. Это эффективно, потому что оно не читает данные из какого-либо временного списка. Оно просто создает итератор, где ко всем производимым строкам применена операция срезания символов.

Для более продвинутого срезания вам стоит обратиться к методу `translate()`. Детали вы найдете в следующем рецепте, где описана чистка строк.

2.12. Чистка строк

Решение

Некий деятель ввел текст “pýthöñ” в форму на вашей веб-странице, и вы хотите как-то почистить эту строку.

Решение

Проблема чистки текста применяется к широкому спектру задач с использованием парсинга текста и обработки данных. На самом элементарном уровне вы можете использовать простые строковые функции (например, `str.upper()` и `str.lower()` для приведения текста к стандартному регистру). Простые замены с использованием `str.replace()` или `re.sub()` помогут справиться с удалением или изменением некоторых специфических последовательностей символов. Вы также можете нормализовать текст, используя функцию `unicodedata.normalize()`, как показано в [рецепте 2.9](#).

Однако вы можете пожелать сделать следующий шаг в процессе чистки. Предположим, например, что вы хотите удалить целые диапазоны символов или удалить диакритические знаки. Для этого вы можете обратиться к методу `str.translate()`. Предположим, у вас есть вот такая замусоренная строка:

```
>>> s = 'pýthöñ\fis\tawesome\r\n'  
>>> s  
'pýthöñ\x0cis\tawesome\r\n'  
>>>
```

Первый шаг — удалить пробел. Сделаем небольшую таблицу перевода и задействуем *translate()*:

```
>>> remap = {
...     ord('\t') : ' ',
...     ord('\f') : ' ',
...     ord('\r') : None # Удалён
...
>>> a = s.translate(remap)
>>> a
'pýthöñ is awesome\n'
>>>
```

Как вы можете увидеть, символы пробелов, такие как \t и \f, были приведены к единой форме. Символ возврата каретки \r был удален.

Вы можете продолжить идею и создать намного более крупные таблицы перевода. Например, давайте удалим все комбинирующиеся символы:

```
>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...     if unicodedata.combining(chr(c)))
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pýthöñ is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>
```

В последнем примере с помощью *dict.fromkeys()* был создан словарь, отображающий все комбинирующиеся символы Unicode на None.

Первоначальные вводные данные затем были нормализованы в декомпозированную форму с использованием *unicodedata.normalize()*. Далее функция *translate()* используется для удаления значков. Похожие приёмы могут быть использованы для удаления символов другого типа (например, управляемых символов).

Ещё один пример — таблица перевода, которая отображает все десятичные цифры Unicode на их эквиваленты в ASCII:

```
>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...     for c in range(sys.maxunicode)
```

```
...     if unicodedata.category(chr(c)) == 'Nd' }
...
>>> len(digitmap)
460
>>> # Арабские цифры
>>> x = '\u0661\u0662\u0663'
>>> x.translate(digitmap)
'123'
>>>
```

Ещё один приём для чистки текста использует функции кодирования и декодирования ввода-вывода. Идея состоит в выполнении некоторой первичной очистки текста, а затем пропускании его через encode() и decode() для срезания символов или изменения. Например:

```
>>> a
'рýтһöñ is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>
```

Здесь процесс нормализации разложил исходный текст на символы вместе с отдельными комбинирующимися символами. Последовательное кодирование и декодирование в ASCII просто удаляет все эти эти символы. Естественно, это сработает только в том случае, если нашей целью было получение ASCII-представления.

Обсуждение

Большой проблемой с чисткой текста может стать производительность. Общее правило: чем проще обработка, тем быстрее она работает. Для простых замен метод str.replace() часто оказывается самым быстрым способом — даже если вызывать его несколько раз. Например, чтобы вычистить пробелы, вы можете использовать такую программу:

```
def clean_spaces(s):
    s = s.replace('\r', '')
    s = s.replace('\t', ' ')
    s = s.replace('\f', ' ')
    return s
```

Если вы попробуете это, то обнаружите, что метод немного быстрее

использования `translate()` или регулярных выражений.

С другой стороны, метод `translate()` очень быстр, если вам нужно выполнить любую нетривиальную операцию замены символов на другие символы или удаления символов.

Производительность — это нечто, что вам придется изучать в каждом конкретном приложении. К сожалению, невозможно предложить один приём, который будет работать лучше всего во всех возможных ситуациях, поэтому пробуйте разные подходы и измеряйте результаты.

Хотя этот рецепт делает акцент на работе с текстом, похожие приёмы могут быть применены к последовательностям байтов.

2.13. Выравнивание текстовых строк

Задача

Вам нужно отформатировать текст с применением некого выравнивания.

Решение

Для базового выравнивания строк можно использовать методы `ljust()`, `rjust()` и `center()`. Например:

```
>>> text = 'Hello World'  
>>> text.ljust(20)  
'Hello World '  
>>> text.rjust(20)  
' Hello World'  
>>> text.center(20)  
' Hello World '  
>>>
```

Все эти методы могут принимать optionalный символ заполнения.

Например:

```
>>> text.rjust(20, '=')  
'=====Hello World'  
>>> text.center(20, '*')  
'*****Hello World*****'  
>>>
```

Функция *format()* также может быть использована для выравнивания. Вам нужно просто использовать символы `<`, `>` или `^` вместе с желаемой шириной. Например:

```
>>> format(text, '>20')
'Hello World'
>>> format(text, '<20')
'Hello World '
>>> format(text, '^20')
'Hello World '
>>>
```

Если вы хотите использовать в качестве заполняющего символа не пробел, определите его перед символом выравнивания:

```
>>> format(text, '=>20s')
'=====Hello World'
>>> format(text, '*^20s')
'*'*'*Hello World*****'
>>>
```

Эти коды форматирования могут быть также использованы с методом *format()* при обработке нескольких значений. Например:

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'Hello World'
>>>
```

У *format()* есть преимущество — он работает не только со строками. Он работает с любыми значениями, что делает его назначение очень широким. Например, вы можете использовать его с числами:

```
>>> x = 1.2345
>>> format(x, '>10')
' 1.2345'
>>> format(x, '^10.2f')
' 1.23 '
>>>
```

Обсуждение

В старых программах вы также можете увидеть, как для форматирования текста использовался оператор `%`. Например:

```
>>> '%-20s' % text
'Hello World '
>>> '%20s' % text
' Hello World'
>>>
```

Однако в новых программах вы должны предпочтать функцию или метод *format()*. Она намного мощнее оператора %. Более того, *format()* может применяться более широко, нежели строковые методы *ljust()*, *rjust()* или *center()*, поскольку работает с любыми объектами.

За полным списком возможностей функции *format()* обратитесь к [документации Python](#).

2.14. Объединение и конкатенация строк

Задача

Вам нужно объединить много небольших строк в длинную строку.

Решение

Если строки, которые вы хотите объединить, находятся в последовательности или итерируемом объекте, самый быстрый способ — использовать метод *join()*. Например:

```
>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ''.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>
```

На первый взгляд синтаксис может показаться странным, однако операция *join()* относится к строковым методам. Объекты, которые вы хотите объединить, могут приходить из разнообразных последовательностей данных (списки, кортежи, словари, файлы, множества или генераторы), поэтому было бы избыточным реализовывать метод *join()* для всех этих объектов. Так что вы просто задаете нужную строку-разделитель, а затем

применяете метод `join()` для склеивания текстовых фрагментов.

Если вы просто объединяете несколько строк, неплохо сработает +:

```
>>> a = 'Is Chicago'  
>>> b = 'Not Chicago?'  
>>> a + ' ' + b  
'Is Chicago Not Chicago?'  
>>>
```

Оператор + также отлично работает в качестве замены более сложным операциям форматирования строк. Например:

```
>>> print('{} {}'.format(a,b))  
Is Chicago Not Chicago?  
>>> print(a + ' ' + b)  
Is Chicago Not Chicago?  
>>>
```

Если вы пытаетесь объединить строковые литералы в исходном коде, вы можете просто разместить их рядом без использования оператора +.

Например:

```
>>> a = 'Hello' 'World'  
>>> a  
'HelloWorld'  
>>>
```

Обсуждение

Объединение строк может показаться недостаточно сложным, чтобы писать про него целый рецепт, но часто эта область является критически важной для производительности.

Важно знать, что использование оператора + для объединения большого количества строк крайне неэффективно, поскольку в памяти создаются копии, что прибавляет работы сборщику мусора. Никогда не пишите такой код для объединения строк:

```
s = ''  
for p in parts:  
    s += p
```

Это работает заметно медленнее метода `join()`, главным образом потому, что каждая `+=` операция создает новый строковый объект. Намного лучше собрать все части и только затем объединить.

Еще один классный фокус из этой области — преобразование данных в строки и конкатенация с одновременным использованием выражения-генератора, как описано в **рецепте 1.19**. Например:

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for d in data)
'ACME,50,91.1'
>>>
```

Берегитесь ненужной конкатенации. Иногда программисты применяют конкатенацию там, где это не нужно. Например:

```
print(a + ':' + b + ':' + c)    # Плохо
print(':' .join([a, b, c]))      # Всё ещё плохо

print(a, b, c, sep=':')         # Лучше
```

Смешивание операций ввода-вывода и конкатенации строк — момент, с которым нужно быть очень внимательными. Например, рассмотрим два фрагмента кода:

```
# Версия 1 (конкатенация строк)
f.write(chunk1 + chunk2)

# Версия 2 (отдельные операции ввода-вывода)
f.write(chunk1)
f.write(chunk2)
```

Если две строки невелики, первая может предложить намного большую производительность (из-за дороговизны системного вызова ввода-вывода). Однако если строки велики, вторая версия может быть более эффективной, поскольку в этом случае не создается огромный промежуточный результат, а также не происходит копирования больших блоков памяти. Пробуйте на своих данных и выясните, что работает быстрее в вашем конкретном случае.

И последнее: если вы пишете код, который формирует результат из множества небольших строк, подумайте о том, чтобы оформить его как генератор, используя `yield` для производства фрагментов. Например:

```
def sample():
    yield 'Is'
    yield 'Chicago'
    yield 'Not'
    yield 'Chicago?'
```

Интересно, что этот подход не делает предположений по поводу того, как фрагменты будут собираться вместе. Например, вы можете просто объединить фрагменты с помощью *join()*:

```
text = ''.join(sample())
```

Или же вы можете перенаправить фрагменты на вывод:

```
for part in sample():
    f.write(part)
```

Или же вы можете создать некую гибридную схему, что умно с точки зрения операций ввода-вывода:

```
def combine(source, maxsize):
    parts = []
    size = 0
    for part in source:
        parts.append(part)
        size += len(part)
        if size > maxsize:
            yield ''.join(parts)
            parts = []
            size = 0
    yield ''.join(parts)

for part in combine(sample(), 32768):
    f.write(part)
```

Ключевой момент в том, что первоначальный генератор не обязан вникать в детали: он просто выдает части.

2.15. Интерполяция переменных в строках

Задача

Вы хотите создать строку, в которой на место переменных будут

подставляться строковые представления значений этих переменных.

Решение

В Python нет прямой поддержки простой подстановки значений переменных в строках. Однако строковый метод *format()* предоставляет приближенную по смыслу возможность: Например:

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

Если значения, которые должны быть подставлены, на самом деле находятся в переменных, вы можете использовать сочетание *format_map()* и *vars()*, как показано тут:

```
>>> name = 'Guido'
>>> n = 37
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

Стоит отметить, что *vars()* также работает с экземплярами. Например:

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
...     def __str__(self):
...         return '{name} has {n} messages.'.format_map(vars(self))
...
>>> a = Info('Guido', 37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

Недостаток *format()* и *format(map)* в том, что они не могут аккуратно справиться с отсутствующими значениями. Например:

```
>>> s.format(name='Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

Этого можно избежать путём определения альтернативного класса словаря с методом `__missing__()`, как показано ниже:

```
class safesub(dict):
    def __missing__(self, key):
        return '{' + key + '}'
```

Теперь этот класс можно использовать, чтобы обернуть значения, которые подаются на вход в `format_map()`:

```
>>> del n # Убедимся, что n не определена
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

Если вы обнаружите, что часто делаете такие вещи в своей программе, вы можете спрятать процесс подстановки переменных в небольшую функцию, которая использует так называемый фреймхак (“frame hack”). Например:

```
import sys
def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

Теперь вы можете делать вот так:

```
>>> name = 'Guido'
>>> n = 37
>>> print(sub('Hello {name}'))
Hello Guido
>>> print(sub('You have {n} messages.'))
You have 37 messages.
>>> print(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>
```

Обсуждение

Отсутствие настоящей интерполяции переменных в Python привело к созданию разнообразных решений. В качестве альтернативы описанным выше, вы иногда можете увидеть такой подход к форматированию строк:

```
>>> name = 'Guido'
>>> n = 37
```

```
>>> '%(name) has %(n) messages.' % vars()
'Guido has 37 messages.'
>>>
```

Или же вам могут попасться строки-темплейты:

```
>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>
```

Однако методы *format()* и *format_map()* являются более современными, нежели эти альтернативы, и отдавать предпочтение нужно им. Преимущество использования *format()* заключается в том, что вы также получаете все возможности форматирования строк (выравнивание, отступы, нумерацию и т.п.), что недоступно для альтернативных решений, таких как строковые объекты *Template*.

В этом рецепте нам также удалось показать несколько интересных продвинутых возможностей. Малоизвестный метод классов словарей и отображений *__missing__()* позволяет вам определить подход для работы с отсутствующими значениями. В классе *safesub* этот метод был определен таким образом, чтобы возвращать отсутствующие значения в форме заглушки (плейсхолдера). Вместо того, чтобы получить исключение *KeyException*, вы увидите отсутствующие значения появляющимися в строке-результате (что может оказаться полезным для отладки).

Функция *sub()* использует *sys._getframe(1)* чтобы вернуть фрейм стека вызывающего. Отсюда атрибут *f_locals* доступен, чтобы получить локальные переменные. Стоит отметить, что заигрывания с фреймами стека стоит избегать. Однако для вспомогательных функций типа строковой подстановки это может оказаться полезным. Отдельно заметим, что *f_locals* — это словарь, который является копией локальных переменных в вызывающей функции. Хотя вы можете изменить содержимое *f_locals*, эти изменения не станут постоянными. Поэтому, хотя доступ другому фрейму стека и может показаться адским злом, невозможно случайно переписать переменные или изменить локальное окружение вызывающей функции.

2.16. Разбивка текста на фиксированное

КОЛИЧЕСТВО КОЛОНOK

Задача

У вас есть длинные строки, которые вы хотите переформатировать таким образом, чтобы они распределились по заданному пользователем количеству колонок.

Решение

Используйте модуль *textwrap* для переформатирования выводимого текста. Предположим, например, что у вас есть такая длинная строка:

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \
the eyes, not around the eyes, don't look around the eyes, \
look into my eyes, you're under."
```

Вот как вы можете использовать модуль *textwrap* чтобы переформатировать её:

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes, don't look around the eyes, look into my eyes,
you're under.

>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, initial_indent=' '))
    Look into my eyes, look into my
eyes, the eyes, the eyes, the eyes, not
around the eyes, don't look around the
eyes, look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, subsequent_indent=' '))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not
around the eyes, don't look around
the eyes, look into my eyes, you're
under.
```

Обсуждение

Модуль *textwrap* — это простой способ очистить текст, особенно если вы хотите, чтобы вывод соответствовал размерам терминала. К вопросу о размере терминала: вы можете получить его, используя `os.get_terminal_size()`. Например:

```
>>> import os  
>>> os.get_terminal_size().columns  
80  
>>>
```

У метода *fill()* есть несколько дополнительных параметров, которые контролируют то, как он обращается с табуляцией, окончаниями предложений и т.д. За подробностями обратитесь к [документации класса `textwrap.TextWrapper`](#).

2.17. Работа с HTML- и XML-сущностями в тексте

Задача

Вы хотите заменить HTML- и XML-сущности, такие как `&entity;` или `&#code;`, соответствующим текстом. Или же вам нужно произвести текст, но экранировать некоторые символы (например, `<`, `>` или `&`).

Решение

Если вы производите текст, довольно просто заменить спецсимволы типа `<` или `>` с помощью функции `html.escape()`. Например:

```
>>> s = 'Elements are written as "<tag>text</tag>".'  
>>> import html  
>>> print(s)  
Elements are written as "<tag>text</tag>".  
>>> print(html.escape(s))  
Elements are written as "“&lt;tag&gt;text&lt;/tag&gt;”.  
  
>>> # Отключим экранирование кавычек  
>>> print(html.escape(s, quote=False))  
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;".
```

```
>>>
```

Если вы хотите произвести текст в кодировке ASCII и вставить коды символов вместо отсутствующих в ASCII символов, вы можете использовать аргумент `errors='xmlcharrefreplace'` с различными функциями ввода-вывода. Например:

```
>>> s = 'Spicy Jalapeño'  
>>> s.encode('ascii', errors='xmlcharrefreplace')  
b'Spicy Jalape&#241;o'  
>>>
```

Чтобы заменить сущности в тексте, нужен другой подход. Если вы обрабатываете HTML или XML, попробуйте для начала настоящий парсер HTML или XML. Обычно эти инструменты автоматически позаботятся о замене значений во время парсинга, и вам не придётся об этом беспокоиться.

Если же по каким-то причинам вы получили голый текст с включением сущностей, и вы хотите заменить их вручную, вы сможете сделать это с помощью различных функций и методов, связанных с парсерами HTML и XML. Например:

```
>>> s = 'Spicy &quot;Jalape&#241;o&quot.'  
>>> from html.parser import HTMLParser  
>>> p = HTMLParser()  
>>> p.unescape(s)  
'Spicy "Jalapeño".'  
>>>  
  
>>> t = '>The prompt is &gt;:&gt;:&gt;'  
>>> from xml.sax.saxutils import unescape  
>>> unescape(t)  
'The prompt is >>>'  
>>>
```

Обсуждение

О правильном экранировании спецсимволов при генерировании HTML или XML легко забыть. Это особенно верно, если вы генерируете вывод самостоятельно, используя `print()` или другую базовую функцию строкового форматирования. Есть простое решение — использовать функции типа `html.escape()`.

Если вам нужно произвести обратное преобразование текста, к вашим услугам различные функции типа `xml.sax.saxutils.unescape()`. Однако мы всё же рекомендуем использовать парсер. Например, если при обработке HTML и XML использовать такие парсеры, как `html.parser` или `xml.etree.ElementTree`, то они самостоятельно позаботятся о замене сущностей в тексте.

2.18. Токенизация текста

Задача

У вас есть строка, которую вы хотите распарсить в поток токенов слева направо.

Решение

Предположим, у вас есть вот такая строка:

```
text = 'foo = 23 + 42 * 10'
```

Чтобы токенизировать строку, вам нужно нечто большее, чем простой поиск по шаблонам. Вам также нужен способ определить тип шаблона. Например, вы можете захотеть превратить строку в последовательность пар:

```
tokens = [ ('NAME', 'foo'), ('EQ', '='), ('NUM', '23'), ('PLUS', '+'),
           ('NUM', '42'), ('TIMES', '*'), ('NUM', '10') ]
```

Для разрезания такого типа первым шагом должно быть определение всех возможных токенов, включая пробелы, с помощью шаблонов регулярных выражений, использующих именованные захватывающие группы:

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ = r'(?P<EQ>=)'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

В этих шаблонах условие `?P\` используется для присваивания имени шаблону.

Это мы используем позже.

Далее, для собственно токенизации, используем малоизвестный метод объектов шаблонов `scanner()`. Этот метод создает объект сканера, в котором повторно вызывается шаг `match()` для предоставленного текста, выполняя один поиск совпадения за раз. Вот интерактивный сеанс работы объекта сканера:

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

Чтобы взять этот приём и использовать в программе, он должен быть очищен и упакован в генератор:

```
from collections import namedtuple

Token = namedtuple('Token', ['type', 'value'])

def generate_tokens(pat, text):
    scanner = pat.scanner(text)
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Пример использования
for tok in generate_tokens(master_pat, 'foo = 42'):
    print(tok)
```

```
# Производит вывод
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

Если вы хотите как-то отфильтровать поток токенов, вы можете либо определить больше генераторов, либо использовать выражение-генератор. Например, вот так можно отфильтровать все токены-пробелы:

```
tokens = (tok for tok in generate_tokens(master_pat, text)
           if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

Обсуждение

Токенизация часто является первым шагом более продвинутого парсинга и обработки текста. Чтобы использовать показанные приёмы сканирования, нужно держать в уме несколько важных моментов. Во-первых, вы должны убедиться, что вы определили соответствующие шаблоны регулярных выражений для всех возможных текстовых последовательностей, которые могут встретиться во входных данных. Если встретится текст, для которого нельзя найти совпадение, сканирование просто остановится. Вот почему необходимо было определить токен пробела (WS) в примере выше.

Порядок токенов в главном регулярном выражении также важен. При поиске совпадений регулярное выражение пытается отыскать совпадения с шаблонами в заданном порядке. Поэтому если шаблон окажется подстрокой более длинного шаблона, вы должны убедиться, что более длинный шаблон вписан в выражение первым. Например:

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Правильно
# master_pat = re.compile('/'.join([LT, LE, EQ])) # Неправильно
```

Второй шаблон неправильный, потому что он будет отыскивать совпадение с `<=`, поскольку за токеном LT следует токен EQ, а не LE.

И последнее: вы должны следить за шаблонами, формирующими подстроки.

Предположим, например, что у вас есть два шаблона:

```
PRINT = r'(P<PRINT>print)'  
NAME  = r'(P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'  
  
master_pat = re.compile('|'.join([PRINT, NAME]))  
  
for tok in generate_tokens(master_pat, 'printer'):  
    print(tok)  
  
# Выводит:  
# Token(type='PRINT', value='print')  
# Token(type='NAME', value='er')
```

Для более продвинутого токенизирования вы можете обратиться к пакетам [PyParsing](#) или [PLY](#). Пример использования PLY вы найдете в следующем рецепте.

2.19. Написание простого парсера на основе метода рекурсивного спуска

Задача

Вам нужно распарсить текст в соответствии с грамматическими правилами и выполнить действия или построить абстрактное синтаксическое дерево, представляющее входные данные.

Решение

В этой задаче мы сосредоточены на парсинге текста в соответствии с некоторой определенной грамматикой. Чтобы это сделать, вы должны начать с формальной спецификации грамматики в форме BNF (БНФ, форма Бэкуса – Наура) или EBNF (РБНФ, расширенная форма Бэкуса – Наура). Например, грамматика для простых арифметических выражений может выглядеть так:

```
expr ::= expr + term  
      | expr - term  
      | term  
  
term ::= term * factor  
       | term / factor
```

```

| factor

factor ::= ( expr )
| NUM

```

А вот альтернативная форма РБНФ:

```

expr ::= term { (+|-) term }*
term ::= factor { (*|/) factor }*
factor ::= ( expr )
| NUM

```

В РБНФ части правил, заключенные в $\{ \dots \}^*$, являются необязательными. $*$ означает ноль и более повторений (то есть имеет такое значение, как и в регулярных выражениях).

Теперь, если вы незнакомы с механизмом работы БНФ, думайте о ней, как об определении правил замены или подстановки, где символы слева могут быть заменены символами справа (или наоборот). В общем, во время парсинга вы пытаетесь сопоставить входящий текст с грамматикой, делая различные подстановки и расширения с использованием БНФ. Чтобы проиллюстрировать это, предположим, что вы парсите выражение $3 + 4 * 5$. Это выражение должно быть сначала разбито на поток токенов с использованием описанных в **рецепте 2.18.** приёмов. Результатом будет последовательность токенов:

NUM + NUM * NUM

С этого момента парсинг начинает пытаться сопоставить грамматику с входящими токенами, делая подстановки:

```

expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*

```

```
expr ::= NUM + NUM * NUM
```

Чтобы пройти по всем шагам подстановки и разобраться, придётся потратить время, но в целом они работают так: смотрят на входящие данные и пытаются сопоставить их с правилами грамматики. Первый входящий токен — это NUM, поэтому подстановки сначала сосредотачиваются на поиске совпадений с этой частью. Когда совпадение найдено, внимание переходит к следующему токену + и т.д. Некоторые части правой стороны (например, { (/ factor }*) исчезают, когда определено, что они не совпадают со следующим токеном. Парсинг проходит успешно, если правая сторона достаточно полна, чтобы охватить все входящие токены.

Со всей вышеизложенной вводной информацией перейдем к простому рецепту построения «выполнителя» выражений, работающего по методу рекурсивного спуска:

```
import re
import collections

# Определение токенов
NUM   = r'(?P<NUM>\d+)'
PLUS  = r'(?P<PLUS>\+)'
MINUS = r'(?P<MINUS>-)'
TIMES = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\())'
RPAREN = r'(?P<RPAREN>\))'
WS    = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))

# Токенизатор
Token = collections.namedtuple('Token', ['type', 'value'])

def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok

# Парсер
class ExpressionEvaluator:
    ...

    Реализация парсера на базе рекурсивного спуска.
    Каждый метод реализует одно правило грамматики.
```

Использует метод `._accept()`, чтобы протестировать и принять текущий токен предварительного просмотра. Использует метод `._expect()` для точного совпадения и отбрасывания следующего токена на входе (или возбуждает `SyntaxError`, если он не совпадает).
'''

```
def parse(self, text):
    self.tokens = generate_tokens(text)

    # Последний потреблённый символ
    self.tok = None
    # Следующий токенизованный символ
    self.nexttok = None
    # Загрузить первый токен предварительного просмотра
    self._advance()

    return self.expr()

def _advance(self):
    'Продвинуться на один токен вперёд'
    self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

def _accept(self, toktype):
    'Проверить и потребить следующий токен, если он совпадает с toktype'
    if self.nexttok and self.nexttok.type == toktype:
        self._advance()
        return True
    else:
        return False

def _expect(self, toktype):
    'Потребить следующий токен, если он совпадает с toktype или возбудить SyntaxError'
    if not self._accept(toktype):
        raise SyntaxError('Expected ' + toktype)

# Далее следуют правила грамматики

def expr(self):
    "expression ::= term { ('+' | '-') term }*"

    exprval = self.term()
    while self._accept('PLUS') or self._accept('MINUS'):
        op = self.tok.type
        right = self.term()
        if op == 'PLUS':
            exprval += right
        elif op == 'MINUS':
            exprval -= right
    return exprval
```

```

def term(self):
    "term ::= factor { ('*' | '/') factor }*"

    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
        if op == 'TIMES':
            termval *= right
        elif op == 'DIVIDE':
            termval /= right
    return termval

def factor(self):
    "factor ::= NUM | ( expr )"

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
        raise SyntaxError('Expected NUMBER or LPAREN')

```

Вот пример интерактивного использования класса ExpressionEvaluator:

```

>>> e = ExpressionEvaluator()
>>> e.parse('2')
2
>>> e.parse('2 + 3')
5
>>> e.parse('2 + 3 * 4')
14
>>> e.parse('2 + (3 + 4) * 5')
37
>>> e.parse('2 + (3 + * 4)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "exprparse.py", line 40, in parse
      return self.expr()
    File "exprparse.py", line 67, in expr
      right = self.term()
    File "exprparse.py", line 77, in term
      termval = self.factor()
    File "exprparse.py", line 93, in factor
      exprval = self.expr()
    File "exprparse.py", line 67, in expr
      right = self.term()

```

```

File "exprparse.py", line 77, in term
    termval = self.factor()
File "exprparse.py", line 97, in factor
    raise SyntaxError("Expected NUMBER or LPAREN")
SyntaxError: Expected NUMBER or LPAREN
>>>

```

Если вы хотите сделать что-то другое, а не только произвести простое вычисление, вам нужно изменить класс ExpressionEvaluator. Например, вот альтернативная реализация, которая конструирует простое дерево разбора (парсинга):

```

class ExpressionTreeBuilder(ExpressionEvaluator):
    def expr(self):
        "expression ::= term { ('+'|'-') term }"

        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval = ('+', exprval, right)
            elif op == 'MINUS':
                exprval = ('-', exprval, right)
        return exprval

    def term(self):
        "term ::= factor { ('*'| '/') factor }"

        termval = self.factor()
        while self._accept('TIMES') or self._accept('DIVIDE'):
            op = self.tok.type
            right = self.factor()
            if op == 'TIMES':
                termval = ('*', termval, right)
            elif op == 'DIVIDE':
                termval = ('/', termval, right)
        return termval

    def factor(self):
        'factor ::= NUM | ( expr )'

        if self._accept('NUM'):
            return int(self.tok.value)
        elif self._accept('LPAREN'):
            exprval = self.expr()
            self._expect('RPAREN')
            return exprval
        else:

```

```
raise SyntaxError('Expected NUMBER or LPAREN')
```

Вот как это работает:

```
>>> e = ExpressionTreeBuilder()
>>> e.parse('2 + 3')
('+', 2, 3)
>>> e.parse('2 + 3 * 4')
('+', 2, ('*', 3, 4))
>>> e.parse('2 + (3 + 4) * 5')
('+', 2, ('*', ('+', 3, 4), 5))
>>> e.parse('2 + 3 + 4')
('+', ('+', 2, 3), 4)
>>>
```

Обсуждение

Парсинг — это обширная тема, освоение которой обычно занимает у студентов первые три недели курса изучения компиляторов. Если вы ищете, где бы почерпнуть знания о грамматиках, алгоритмах разбора и прочую подобную информацию, обратитесь к книгам о компиляторах. Нет нужды говорить, что всё это втиснуть в данную книгу просто невозможно.

Тем не менее, общая идея парсера на основе рекурсивного спуска проста. Для начала вы берете каждое правило грамматики и превращаете его в функцию или метод. Если ваша грамматика выглядит так:

```
expr ::= term { ('+'|'-') term }*
term ::= factor { ('*'|'/') factor }*
factor ::= '(' expr ')'
| NUM
```

...то вы начинаете с превращения её в такой набор методов:

```
class ExpressionEvaluator:
    ...
    def expr(self):
        ...
    def term(self):
        ...
    def factor(self):
        ...
```

Задача каждого метода проста: он должен пройти слева направо по каждой

части грамматического правила, потребляя токены в процессе. Цель метода — либо потребить правило, либо сгенерировать синтаксическую ошибку в случае застревания. Чтобы реализовать это, применяются следующие приёмы:

- Если следующий символ в правиле является именем другого грамматического правила (например, `term` или `factor`), вы просто вызываете метод с этим именем. Это «спуск» алгоритма — управление спускается в другое грамматическое правило. Иногда правила могут использовать вызовы методов, которые уже выполняются (например, вызов `expr` в правиле `factor ::= '(' expr ')'.`). Это «рекурсивность» алгоритма.
- Если следующий символ в правиле должен быть конкретным символом (например, `()`, вы смотрите на следующий токен и проверяете на точное совпадение. Если он не совпадает, то это синтаксическая ошибка. В этом рецепте для выполнения этих шагов используется метод `_expect()`.
- Если следующий символ в правиле может соответствовать нескольким возможным выборам (например, `+` или `-`), вы должны проверить следующий токен на каждую из этих возможностей и продвигаться вперед только в том случае, если совпадение найдено. В этом рецепте за это отвечает метод `_accept()`. Он похож на более слабую версию метода `_expect()` — в том отношении, что он продвинется вперед, только если совпадение найдено, но если нет, то он просто отступает, не возбуждая ошибку (что позволяет сделать другие проверки).
- Для грамматических правил с повторяющимися частями (как, например, в правиле `expr ::= term { ('+'|'-') term }*`), повторение реализуется циклом `while`. Тело цикла будет в общем собирать или обрабатывать все повторяющиеся значения, пока они не закончатся.
- Если грамматическое правило потреблено, каждый метод возвращает некий результат тому, кто его вызывал. Так значения распространяются во время парсинга. Например, в «вычислителе» выражений возвращаемые значения будут представлять частичные результаты разбираемого выражения. В конце концов они все объединяются в высшем методе грамматического правила, который будет выполнен.

Хотя здесь мы показали простой пример, парсеры на основе рекурсивного спуска могут быть использованы для создания весьма сложных парсеров. Например, код самого Python интерпретируется парсером на основе метода рекурсивного спуска. Если вы заинтересовались, вы можете залезть в файл `Grammar/Grammar` в исходном коде Python и взглянуть на лежащую в основе

грамматику. При всём при этом, конечно, в ручном создании парсеров множество ограничений и ловушек.

Одно из таких ограничений парсеров на основе рекурсивного спуска заключается в том, что они не могут быть написаны для грамматических правил, использующих левую рекурсию. Предположим, например, что вам нужно перевести такое правило:

```
items ::= items ',' item
| item
```

Чтобы сделать это, вы могли бы использовать метод *items()*:

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
        itemsval.append(self.item())
    else:
        itemsval = [ self.item() ]
```

Единственная проблема в том, что это не работает. Такой код вылетит с ошибкой бесконечной рекурсии.

Вы можете также столкнуться с хитрыми проблемами, касающимися самих грамматических правил. Например, вы можете поразмышлять над тем, могут ли выражения быть описаны вот такой более простой грамматикой:

```
expr ::= factor { ('+' | '-' | '*' | '/') factor }*
factor ::= '(' expression ')'
| NUM
```

Эта грамматика технически «работает», но она не соблюдает стандартные правила порядка вычисления арифметических выражений. Например, для выражения “3 + 4 * 5” оно выдаст результат 35 вместо правильного 23. Чтобы решить эту проблему, нужно использовать отдельные правила *expr* и *term*.

Для по-настоящему сложных грамматик лучше использовать инструменты парсинга типа [PyParsing](#) или [PLY](#). Вот как выглядит код «вычислителя» выражений, созданный с применением PLY:

```
from ply.lex import lex
from ply.yacc import yacc
```

```

# Список токенов
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN'

# Игнорируемые символы
t_ignore = ' \t\n'

# Определения токенов (в форме регулярных выражений)
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Функции обработки токенов
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Обработчик ошибок
def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Создание лексера
lexer = lex()

# Правила грамматики и функции-обработчики
def p_expr(p):
    '''
    expr : expr PLUS term
          | expr MINUS term
    ...
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]

def p_expr_term(p):
    ...
    expr : term
    ...
    p[0] = p[1]

def p_term(p):
    ...
    term : term TIMES factor
          | term DIVIDE factor

```

```

...
if p[2] == '*':
    p[0] = p[1] * p[3]
elif p[2] == '/':
    p[0] = p[1] / p[3]

def p_term_factor(p):
    ...
    term : factor
    ...
    p[0] = p[1]

def p_factor(p):
    ...
    factor : NUM
    ...
    p[0] = p[1]

def p_factor_group(p):
    ...
    factor : LPAREN expr RPAREN
    ...
    p[0] = p[2]

def p_error(p):
    print('Syntax error')

parser = yacc()

```

В этой программе вы обнаружите, что всё определено так же, как и ранее написанном парсере, но на намного более высоком уровне. Вы просто пишете регулярные выражения для токенов и высокоуровневые функции-обработчики, которые выполняются, когда возникают совпадения по различным правилам грамматики. А вся механика работы парсера, приёма токенов и так далее полностью реализована в библиотеке.

Вот пример использования созданного объекта парсера:

```

>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>

```

Если вы хотите сделать свою программерскую жизнь более захватывающей, начните писать парсеры и компиляторы. Повторимся, книги про компиляторы предлагают кучу низкоуровневых подробностей и тонны теории. Множество полезных ресурсов и всякой информации вы также найдете в сети. А в Python есть модуль *ast*, на который также стоит посмотреть.

2.20. Выполнение текстовых операций над байтовыми строками

Задача

Вы хотите выполнить стандартные текстовые операции (срезание символов, поиск, замену) над строками байтов.

Решение

Байтовые строки поддерживают большую часть тех же встроенных операций, что и текстовые строки. Например:

```
>>> data = b'Hello World'  
>>> data[0:5]  
b'Hello'  
>>> data.startswith(b'Hello')  
True  
>>> data.split()  
[b'Hello', b'World']  
>>> data.replace(b'Hello', b'Hello Cruel')  
b'Hello Cruel World'  
>>>
```

Такие операции можно проделать и над байтовыми массивами:

```
>>> data = bytearray(b'Hello World')  
>>> data[0:5]  
bytearray(b'Hello')  
>>> data.startswith(b'Hello')  
True  
>>> data.split()  
[bytearray(b'Hello'), bytearray(b'World')]  
>>> data.replace(b'Hello', b'Hello Cruel')  
bytearray(b'Hello Cruel World')  
>>>
```

Вы можете просто применить к байтовым строкам поиск совпадений с помощью регулярных выражений, но сами шаблоны должны быть определены как байты. Например:

```
>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re
>>> re.split('[:,]',data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python3.3/re.py", line 191, in split
      return _compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object

>>> re.split(b'[:,]',data) # Обратите внимание: шаблон в байтах
[b'FOO', b'BAR', b'SPAM']
>>>
```

Обсуждение

Практически все доступные для текстовых строк операции будут работать и на байтовых строках. Однако есть несколько заметных отличий, о которых нужно знать. Во-первых, при индексировании байтовых строк мы получаем целые числа, а не символы. Например:

```
>>> a = 'Hello World' # Текстовая строка
>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World' # Байтова строка
>>> b[0]
72
>>> b[1]
101
>>>
```

Эта разница в семантике может влиять на программы, которые пытаются обработать байтовые данные так же, как и текстовые.

Во-вторых, байтовые строки не предоставляют красивые строковые представления и не выводятся в симпатичном виде, если сначала не проведено декодирование в текстовую строку. Например:

```
>>> s = b'Hello World'
>>> print(s)
b'Hello World' # Обратите внимание на b'...'
>>> print(s.decode('ascii'))
Hello World
>>>
```

Строковые операции форматирования также недоступны для байтовых строк.

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'

>>> b'{ } { } {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

Если вы хотите применить какое-то форматирование к байтовой строке, это должно быть проделано с помощью обычных текстовых строк и последующего кодирования. Например:

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')
b'ACME 100 490.10'
>>>
```

И, наконец, вы должны знать, что использование байтовых строк может изменить семантику некоторых операций — особенно тех, что относятся к файловой системе. Например, если вы предоставляетете имя файла закодированным в байтовую строку, а не в текстовую, это обычно отключает кодирование и декодирование имени файла. Например:

```
>>> # Запишем имя файла в UTF-8
>>> with open('jalape\xflo.txt', 'w') as f:
...     f.write('spicy')
...
>>> # Получим содержимое каталога
>>> import os
>>> os.listdir('.') # Текстовая строка (имена декодированы)
['jalapeño.txt']
```

```
>>> os.listdir(b'.') # Байтовая строка (имена остались байтами)
[b'jalapen\xcc\x83o.txt']
>>>
```

Посмотрите, как в последней части этого примера передача имени каталога в виде байтовой строки вызывает возврат имен файлов в виде недекодированных байтов. Имя файла, показанное в списке содержимого каталога, содержит «сырую» кодировку UTF-8. См. [рецепт 5.15.](#), в нем обсуждается вопрос работы с именами файлов, имеющий отношение к этому случаю.

Некоторые программисты могут склоняться к использованию байтовых строк в качестве альтернативы текстовым из-за возможного выигрыша в производительности. Да, операции над байтами могут быть немного более эффективными, чем работа с текстом (из-за оверхеда на Unicode), однако такой подход приводит к грязному и неидиоматическому коду. Вы будете часто сталкиваться с тем, что байтовые строки не очень хорошо сочетаются с другими частями Python, и закончите тем, что будете вручную выполнять всевозможные операции кодирования-декодирования, чтобы всё работало. Так что если вы работаете с текстом, используйте обычные текстовые строки, а не байтовые.

3. Числа, даты и время

В Python легко выполнять математические вычисления с целыми числами и числами с плавающей точкой. Однако если вам нужно работать с дробями, массивами или датами и временем, придется приложить больше усилий. Эта глава фокусируется как раз на таких темах.

3.1. Округление числовых значений

Задача

Вы хотите округлить число с плавающей точкой до заданного количества знаков после точки.

Решение

Для простого округления используйте встроенную функцию `round(value, ndigits)`. Например:

```
>>> round(1.23, 1)
1.2
>>> round(1.27, 1)
1.3
>>> round(-1.27, 1)
-1.3
>>> round(1.25361, 3)
1.254
>>>
```

Когда значение попадает точно между двух возможных выборов для округления, эта функция будет округлять к ближайшему чётному значению. То есть 1.5 или 2.5 будут округлены до 2.

Количество знаков, которое передается функции `round()`, может быть отрицательным. В этом случае округление будет идти до десятков, сотен, тысяч и т.д. Например:

```
>>> a = 1627731
>>> round(a, -1)
1627730
>>> round(a, -2)
1627700
>>> round(a, -3)
1628000
>>>
```

Обсуждение

Не перепутайте округление с форматированием значения для вывода. Если вы хотите просто вывести число с некоторым определенным количеством знаков после точки, обычно вам не требуется `round()`. Вместо этого просто задайте при форматировании, сколько знаков выводить. Пример:

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:.3f}'.format(x)
'value is 1.235'
```

```
>>>
```

Сопротивляйтесь желанию округлить числа с плавающей точкой, чтобы исправить проблемы с точностью вычислений. Например, вы можете подумывать поступить так:

```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.300000000000001
>>> c = round(c, 2) # "Исправленный" результат (??)
>>> c
6.3
>>>
```

Для большинства программ, работающих с числами с плавающей точкой, просто не нужно (и не рекомендуется) этого делать. Хотя есть незначительные ошибки в вычислениях, поведение этих ошибок понятно и терпимо. Если необходимо избежать таких ошибок (например, это может быть важно для финансовых приложений), попробуйте модуль *decimal*, который обсуждается в следующем рецепте.

3.2. Выполнение точных вычислений с десятичными дробями

Задача

Вам нужно выполнить точные вычисления с десятичными дробями, и вы хотите избавиться от небольших ошибок, которые обычно возникают при работе с числами с плавающей точкой.

Решение

Широко известный недостаток чисел с плавающей точкой в том, что они не могут точно представить все 10 базовых десятичных цифр. Более того, даже простые математические вычисления приводят к появлению небольших ошибок. Например:

```
>>> a = 4.2
```

```
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a + b) == 6.3
False
>>>
```

Эти ошибки — не «бага, а фича» процессора и стандарта представления чисел с плавающей точкой IEEE 754, на основе которого работает модуль процессора для вычислений с плавающей точкой. Поскольку в типе данных «числа с плавающей точкой» Python хранит данные, используя нативное представление, вы ничего не можете сделать, чтобы избавиться от ошибок при использовании экземпляров *float*.

Если вам нужна большая точность (и вы готовы в некоторой степени поступиться производительностью), вы можете использовать модуль *decimal*:

```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
>>>
```

На первый взгляд он может показаться странным (например, определение чисел как строк). Однако объекты *Decimal* работают именно так, как вы можете ожидать (поддерживают все обычные математические операции и т.д.) Если вы выводите их или используете функциях форматирования строк, они выглядят как обычные числа.

Главное преимущество *decimal* в том, что он позволяет контролировать различные аспекты вычислений, такие как число знаков после точки и округление. Чтобы это сделать, вы создаете локальный контекст и меняете его установки. Например:

```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.7647058823529411764705882353
>>> with localcontext() as ctx:
```

```
...     ctx.prec = 3
...     print(a / b)
...
0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>
```

Обсуждение

Модуль *decimal* реализует «Общую спецификацию десятичной арифметики» компании IBM (“General Decimal Arithmetic Specification”). Нет нужды упоминать, что у него есть очень много различных опций для конфигурирования, описание которых лежит за пределами возможностей этой книги.

Новички в Python могут склоняться к повсеместному использованию модуля *decimal* для решения проблемы неточности, которая неизбежна при работе с типом данных *float*. Однако важно понимать область применения вашего приложения. Если вы работаете с научными или инженерными данными, компьютерной графикой, то вполне нормально использовать обычный тип данных чисел с плавающей точкой. В общем-то очень немногие вещи в реальном мире измеряются с точностью до 17-го знака после точки, которую предоставляет *float*. Так что небольшие ошибки не так уж важны. А производительность нативных чисел с плавающей точкой заметно выше, что важно при выполнении большого количества вычислений.

Но вы не должны просто полностью игнорировать ошибки. Математики проводят немало времени, изучая различные алгоритмы, и некоторые обрабатывают ошибки лучше других. Вы также должны быть осторожными с эффектами таких штук, как вычитательная потеря точности и сложение больших и маленьких чисел. Например:

```
>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums) # Заметьте, как исчезает 1
0.0
>>>
```

Ошибка из последнего примера может быть решена путем использования

math.fsum():

```
>>> import math  
>>> math.fsum(nums)  
1.0  
>>>
```

Однако для других алгоритмов вам придется изучить реализацию и понять, как они работают с точки зрения подобных ошибок.

Подведем итог: модуль *decimal* используется в основном в финансовых и прочих подобных приложениях. В таких программах небольшие ошибки в вычислениях ужасно мешают, а *decimal* позволяет от них избавиться. Также часто можно встретить объекты класса *Decimal* в интерфейсах Python к базам данных — опять же, особенно часто их используют для доступа к финансовым данным.

3.3. Форматирование чисел для вывода

Задача

Вам нужно отформатировать число для вывода, контролируя количество знаков, выравнивание, включение разделителя для разрядов и т.д.

Решение

Чтобы отформатировать одно число для вывода, используйте встроенную функцию *format()*. Например:

```
>>> x = 1234.56789  
  
>>> # Два десятичных знака точности  
>>> format(x, '0.2f')  
'1234.57'  
  
>>> # Выравнивание по правому краю в 10 символов, один знак точности  
>>> format(x, '>10.1f')  
' 1234.6'  
  
>>> # Выравнивание по левому краю  
>>> format(x, '<10.1f')  
'1234.6 '
```

```
>>> # Выравнивание по центру
>>> format(x, '^10.1f')
' 1234.6 '

>>> # Включение разделителя разрядов
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
>>>
```

Если вы хотите использовать экспоненциальную нотацию, измените f на e или E (в зависимости от регистра, который вы хотите использовать для обозначения экспоненты). Например:

```
>>> format(x, 'e')
'1.234568e+03'
>>> format(x, '0.2E')
'1.23E+03'
>>>
```

Общая форма ширины и точности в обоих случаях такова: '[<>^]?width[,]?(.digits)?', где width и digits — целые числа, а ? обозначает необязательные части. Тот же формат используется в строковом методе *format()*. Например:

```
>>> 'The value is {:.2f}'.format(x)
'The value is 1,234.57'
>>>
```

Обсуждение

Форматирование чисел для вывода обычно вполне бесхитростно. Приём, показанный выше, работает и для чисел с плавающей точкой, и для экземпляров *Decimal* из модуля *decimal*.

Когда количество знаков ограничено, значения округляются таким же образом, как и при использовании функции *round()*. Например:

```
>>> x
1234.56789
>>> format(x, '0.1f')
'1234.6'
>>> format(-x, '0.1f')
```

```
'-1234.6'  
>>>
```

Обычное форматирование значений с добавлением разделителя разрядов ничего не знает о принятых в конкретных странах традициях форматирования тысячных разрядов. Если вам нужно принять во внимание эти традиции, обратите внимание на функции модуля *locale*. Вы также можете заменить символ разделителя разрядов, используя строковый метод *translate()*. Например:

```
>>> swap_separators = { ord('.'): ',', ord(','):'.' }  
>>> format(x, ',').translate(swap_separators)  
'1.234,56789'  
>>>
```

В мире всё ещё очень много кода, использующего форматирование чисел на основе оператора %. Например:

```
>>> '%0.2f' % x  
'1234.57'  
>>> '%10.1f' % x  
' 1234.6'  
>>> '%-10.1f' % x  
'1234.6 '  
>>>
```

Это форматирование всё еще приемлемо, но обладает меньшими возможностями, нежели современный метод *format()*. Например, форматирование с помощью оператора % не поддерживает добавление разделителя разрядов.

3.4. Работа с бинарными, восьмеричными и шестнадцатеричными целыми числами

Задача

Вам нужно преобразовать выводимые целые числа в бинарное, восьмеричное или шестнадцатеричное представление.

Решение

Чтобы преобразовать целое число в бинарное, восьмеричное или шестнадцатеричное представление, используйте функции *bin()*, *oct()* или *hex()* соответственно:

```
>>> x = 1234
>>> bin(x)
'0b10011010010'
>>> oct(x)
'0o2322'
>>> hex(x)
'0x4d2'
>>>
```

Или же вы можете использовать функцию *format()*, если не хотите, чтобы появлялись префиксы 0b, 0o или 0x. Например:

```
>>> format(x, 'b')
'10011010010'
>>> format(x, 'o')
'2322'
>>> format(x, 'x')
'4d2'
>>>
```

Целые числа имеют знак, поэтому если вы работаете с отрицательными значениями, то вывод также будет включать знак. Например:

```
>>> x = -1234
>>> format(x, 'b')
'-10011010010'
>>> format(x, 'x')
'-4d2'
>>>
```

Если вы хотите вывести значение без знака, вам нужно добавить максимальное значение, чтобы установить длину бита. Например, чтобы вывести 32-битное значение, можно поступить так:

```
>>> x = -1234
>>> format(2**32 + x, 'b')
'111111111111111111111101100101110'
>>> format(2**32 + x, 'x')
'fffffb2e'
>>>
```

Чтобы преобразовать строки с целыми числами в числа с разными основаниями, используйте функцию `int()`, указав нужное основание. Например:

```
>>> int('4d2', 16)
1234
>>> int('10011010010', 2)
1234
>>>
```

Обсуждение

По большей части работа с бинарными, восьмеричными и шестнадцатеричными целыми числами прямолинейна. Просто запомните, что эти преобразования относятся только выводу разных текстовых представлений чисел. «Под капотом» это один и тот же тип целых чисел.

Предупреждение для программистов, работающих с восьмеричными числами: синтаксис Python для определения восьмеричных значений немного отличается от реализованного в большинстве других языков. Если вы попробуете сделать это так, то получите синтаксическую ошибку:

```
>>> import os
>>> os.chmod('script.py', 0755)
  File "<stdin>", line 1
    os.chmod('script.py', 0755)
                           ^
SyntaxError: invalid token
>>
```

Убедитесь, что вы вводите восьмеричное значение с префиксом 0о, как показано тут:

```
>>> os.chmod('script.py', 0o755)
>>>
```

3.5. Упаковка и распаковка больших целых чисел из байтовых строк

Задача

У вас есть строка байтов, и вам нужно распаковать ее в целочисленное значение. Или же вам нужно конвертировать большое целое число в байтовую строку.

Решение

Предположим, ваша программа должна работать с байтовой строкой из 16 элементов, которая содержит 128-битное целочисленное значение.

Например:

```
data = b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

Чтобы перевести байты в целое число, используйте `int.from_bytes()`, определив порядок следования байтов таким образом:

```
>>> len(data)
16
>>> int.from_bytes(data, 'little')
69120565665751139577663547927094891008
>>> int.from_bytes(data, 'big')
94522842520747284487117727783387188
>>>
```

Чтобы преобразовать большое целочисленное значение обратно в байтовую строку, используйте метод `int.to_bytes()`, определив количество байтов и порядок их следования. Например:

```
>>> x = 94522842520747284487117727783387188
>>> x.to_bytes(16, 'big')
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> x.to_bytes(16, 'little')
b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00v4\x12\x00'
>>>
```

Обсуждение

Преобразование больших целочисленных значений из и в байтовые строки — не самая обычная операция. Однако иногда такая задача возникает в областях типа криптографии или работы с сетью. Например, сетевые адреса IPv6 представлены 128-битными целыми числами. Если вы пишете программу, в которой нужно вытягивать такие значения из данных, вы можете столкнуться с этой задачей.

В качестве альтернативы вы можете попытаться распаковывать значения, используя модуль *struct*, как описано в [рецепте 6.11](#). Это работает, но размер целых чисел, которые могут быть распакованы с помощью *struct*, ограничен. Поэтому вам понадобится распаковывать несколько значений и объединять их для создания итогового значения. Например:

```
>>> data
b'\x00\x124V\x00\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> import struct
>>> hi, lo = struct.unpack('>QQ', data)
>>> (hi << 64) + lo
94522842520747284487117727783387188
>>>
```

Определение порядка следования байтов (*little* или *big*), просто указывает, записаны ли байты, из которых составляется целое число, в порядке от старшего к младшему или наоборот. Это легко понять, рассмотрев пример такого специально составленного шестнадцатеричного значения:

```
>>> x = 0x01020304
>>> x.to_bytes(4, 'big')
b'\x01\x02\x03\x04'
>>> x.to_bytes(4, 'little')
b'\x04\x03\x02\x01'
>>>
```

Если вы хотите упаковать целое число в строку байтов, но оно не поместится, вы получите ошибку. При необходимости вы можете использовать метод *int.bit_length()*, чтобы определить, сколько байтов потребуется для хранения значения:

```
>>> x = 523 ** 23
>>> x
335381300113661875107536852714019056160355655333978849017944067
>>> x.to_bytes(16, 'little')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> x.bit_length()
208
>>> nbytes, rem = divmod(x.bit_length(), 8)
>>> if rem:
...     nbytes += 1
...
>>>
```

```
>>> x.to_bytes(nbytes, 'little')
b'\x03\x1f\x82iT\x96\xac\xc7c\x16\xf3\xb9\xcf...\xd0'
>>>
```

3.6. Вычисления с комплексными числами

Задача

Возможно, ваша программа для взаимодействия с веб-сервисом для аутентификации последнего поколения столкнулась с сингулярностью, и ваш единственный способ обойти это лежит через комплексную плоскость... Или же вам просто нужно выполнить какие-то вычисления с использованием комплексных чисел.

Решение

Комплексные числа могут быть определены с использованием функции `complex(real, imag)` или добавлением окончания `j` к числу с плавающей точкой. Например:

```
>>> a = complex(2, 4)
>>> b = 3 - 5j
>>> a
(2+4j)
>>> b
(3-5j)
>>>
```

Реальное, мнимое и объединенное значения получить легко:

```
>>> a.real
2.0
>>> a.imag
4.0
>>> a.conjugate()
(2-4j)
>>>
```

Работают все обычные математические операторы:

```
>>> a + b
(5-1j)
>>> a * b
(26+2j)
>>> a / b
(-0.4117647058823529+0.6470588235294118j)
>>> abs(a)
4.47213595499958
>>>
```

Для специальных операций с комплексными числами, таких как синусы, косинусы или квадратные корни, используйте модуль *cmath*:

```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```

Обсуждение

Большинство связанных с математикой модулей Python умеют работать с комплексными числами. Например, если вы используете *numpy*, то сможете создавать массивы комплексных чисел и выполнять операции над ними:

```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j,  4.+5.j,  6.-7.j,  8.+9.j])
>>> a + 2
array([ 4.+3.j,  6.+5.j,  8.-7.j, 10.+9.j])
>>> np.sin(a)
array([ 9.15449915 -4.16890696j, -56.16227422 -48.50245524j,
-153.20827755-526.47684926j, 4008.42651446-589.49948373j])
>>>
```

Стандартные математические функции, включенные в Python, не производят комплексные значения по умолчанию, так что они вряд ли случайно возникнут в вашем коде. Например:

```
>>> import math
>>> math.sqrt(-1)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

Если вы хотите получать в результате вычислений комплексные числа, вы должны явно использовать `cmath` или соответствующим образом объявить это библиотекам, которые умеют с ними работать. Например:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
>>>
```

3.7. Работа с бесконечными значениями и NaN

Задача

Вам нужно создать или протестировать такие значения с плавающей точкой: бесконечность, минус бесконечность, NaN (not a number, «не число»).

Решение

В Python нет специального синтаксиса для представления таких специальных значений с плавающей точкой, но они могут быть созданы с помощью `float()`. Например:

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

Чтобы проверить, не является ли значение таким, используйте функции `math.isinf()` и `math.isnan()`. Например:

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

Обсуждение

За подробностями об этих специальных значениях с плавающей точкой вы можете обратиться к спецификации IEEE 754. Однако здесь есть несколько хитрых деталей, о которых нужно знать. Особенное внимание нужно обратить на темы, связанные со сравнениями и операторами.

Бесконечные значения распространяются в вычислениях согласно математическим правилам. Например:

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
>>>
```

Однако некоторые операции неопределены и выдают NaN. Например:

```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
>>>
```

Значения NaN распространяются через все операции, не возбуждая исключений. Например:

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
```

```
>>> math.sqrt(c)
nan
>>>
```

Тонкость с NaN заключается в том, что они никогда будут равны друг другу. Например:

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
>>>
```

По причине этого единственный безопасный способ проверить значение на NaN — это использовать *math.isnan()*, как показано в этом рецепте.

Иногда программисты хотят изменить поведение Python таким образом, чтобы при возникновении в ходе вычислений бесконечностей или NaN возбуждались исключения. Для такого изменения поведения может быть использован модуль *fpectl*, но он не включен в стандартную поставку Python, является платформозависимым и на самом деле предназначен только для программистов-экспертов. За деталями обратитесь к [онлайн-документации Python](#).

3.8. Вычисления с дробями

Задача

Вы вошли в машину времени и внезапно обнаружили себя делающим домашку по математике с задачками про дроби. Или же вы просто пишете код, который будет обсчитывать измерения, сделанные в вашей столярной мастерской...

Решение

Модуль *fractions* может быть использован для выполнения математических операций с дробями. Например:

```
>>> from fractions import Fraction
```

```
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Получение числителя/знаменателя
>>> c = a * b
>>> c.numerator
35
>>> c.denominator
64

>>> # Converting to a float
>>> float(c)
0.546875
>>> # Ограничиваем знаменатель значения
>>> print(c.limit_denominator(8))
4/7

>>> # Конвертируем float в дробь
>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

Обсуждение

Вычисления с дробями нечасто возникают в обычных программах, но иногда имеет смысл ими воспользоваться. Например, если данные каких-то измерений поступают в виде дробей, то можно работать прямо с ними, что снимает необходимость конвертирования в десятичные дроби или числа с плавающей точкой.

3.9. Вычисления на больших массивах чисел

Задача

Вам нужно произвести вычисления на больших объемах числовых данных, таких как массивы или решетки.

Решение

Для любых объемных вычислений с использованием массивов используйте библиотеку [NumPy](#). Ее главное преимущество в том, что она предоставляет Python объект массива, который намного эффективнее и лучше подходит для математических вычислений, нежели стандартный список Python. Вот короткий пример, иллюстрирующий важные различия между обычными списками и массивами NumPy:

```
>>> # Python lists
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # массивы NumPy
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6,  8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

Как вы можете видеть, базовые математические операции с использованием массивов выполняются по-разному. Конкретно скалярные операции (например, $ax * 2$ или $ax + 10$) применяют операцию элемент за элементом. Также отметим, что выполнение таких математических операций, где каждый из операндов является массивом, применяет операцию ко всем элементам и создает новый массив.

Тот факт, что математические операции применяются одновременно ко всем элементам, позволяет очень просто и быстро применить функции к всему массиву. Например, если вы хотите вычислить значение многочлена:

```
>>> def f(x):
...     return 3*x**2 - 2*x + 7
...
>>> f(ax)
array([ 8, 15, 28, 47])
>>>
```

NumPy предоставляет набор «универсальных функций», которые также работают для операций над массивами. Они подменяют похожие функции, доступные в модуле *math*. Например:

```
>>> np.sqrt(ax)
array([ 1. , 1.41421356, 1.73205081, 2. ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>>
```

Использование универсальных функций позволяет выполнить вычисление в сотни раз быстрее, чем проход по массиву и применение функций из *math* к каждому элементу. Так что используйте их при любой возможности.

«Под капотом» массивы NumPy устроены похожим на массивы C или Fortran образом. А именно, они представляют собой большие смежные области памяти, состоящие из однородных типов данных. Это дает возможность делать массивы намного более крупными, чем позволяет обычный список Python. Например, если вы хотите создать двумерную решетку размером 10 000 на 10 000 чисел с плавающей точкой, это не проблема:

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
>>>
```

Все обычные операции всё ещё применяются к элементам одновременно:

```
>>> grid += 10
>>> grid
array([[ 10., 10., 10., ..., 10., 10., 10.],
```

```

[ 10., 10., 10., ..., 10., 10., 10.],
[ 10., 10., 10., ..., 10., 10., 10.],
...,
[ 10., 10., 10., ..., 10., 10., 10.],
[ 10., 10., 10., ..., 10., 10., 10.],
[ 10., 10., 10., ..., 10., 10., 10.])
>>> np.sin(grid)
array([[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
       -0.54402111, -0.54402111],
      [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
       -0.54402111, -0.54402111],
      [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
       -0.54402111, -0.54402111],
      ...,
      [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
       -0.54402111, -0.54402111],
      [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
       -0.54402111, -0.54402111],
      [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
       -0.54402111, -0.54402111]]))
>>>

```

Важнейший момент в использовании NumPy — это способ, которым она расширяет функциональность индексирования списков Python (особенно для многомерных массивов). Чтобы проиллюстрировать это, создадим простой двумерный массив и поэкспериментируем:

```

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> # Выбираем строку 1
>>> a[1]
array([5, 6, 7, 8])

>>> # Выбираем колонку 1
>>> a[:,1]
array([ 2,  6, 10])

>>> # Выбираем и изменяем субрегион
>>> a[1:3, 1:3]
array([[ 6,  7],
       [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 10, 11, 12]])

```

```
[ 9, 20, 21, 12]])

>>> # Транслируем строковый вектор на операции со всеми строками
>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
       [105, 117, 119, 111],
       [109, 121, 123, 115]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])

>>> # Условное присваивание в массиве
>>> np.where(a < 10, a, 10)
array([[ 1,  2,  3,  4],
       [ 5, 10, 10,  8],
       [ 9, 10, 10, 10]])
>>>
```

Обсуждение

NumPy — это основа огромного количества научных и инженерных библиотек для Python. Это также один из крупнейших и самых сложных модулей (из тех, что широко используются). При этом можно делать полезные вещи с помощью NumPy, начав экспериментировать с простыми примерами.

Стоит отметить, что часто используется конструкция *import numpy as np*, как и показано в нашем примере. Это сокращает название, чтобы было удобно вводить его снова и снова в вашей программе.

Прочую информацию вы найдёте на <http://www.numpy.org>.

3.10. Вычисления с матрицами и линейная алгебра

Задача

Вам нужно произвести матричные операции и операции линейной алгебры, такие как умножение матриц, поиск определителей, решение линейных уравнений и т.д.

Решение

Библиотека [NumPy](#) содержит объект *matrix*. Матрицы — это нечто похожее на объекты массивов, описанные в [рецепте 3.9.](#), но в вычисления над ними следуют законам линейной алгебры. Вот несколько примеров их основных возможностей:

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2,  3],
        [ 0,  4,  5],
        [ 7,  8, -9]])

>>> # Возвращает транспонированную
>>> m.T
matrix([[ 1,  0,  7],
        [-2,  4,  8],
        [ 3,  5, -9]])

>>> # Возвращает инвертированную
>>> m.I
matrix([[ 0.33043478, -0.02608696,  0.09565217],
        [-0.15217391,  0.13043478,  0.02173913],
        [ 0.12173913,  0.09565217, -0.0173913 ]])

>>> # Создаем вектор и умножаем
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[ 2],
        [ 3],
        [ 4]])

>>> m * v
matrix([[ 8],
        [32],
        [ 2]])

>>>
```

Другие операции можно найти в субпакете *numpy.linalg*. Например:

```
>>> import numpy.linalg
>>> # Детерминант
>>> numpy.linalg.det(m)
-229.9999999999983

>>> # Собственные значения
>>> numpy.linalg.eigvals(m)
```

```
array([-13.11474312, 2.75956154, 6.35518158])

>>> # Решение для x в mx = v
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
       [ 0.17391304],
       [ 0.46086957]])
>>> m * x
matrix([[ 2.],
       [ 3.],
       [ 4.]])
>>> v
matrix([[ 2.],
       [ 3.],
       [ 4.]])
>>>
```

Обсуждение

Линейная алгебра, очевидно, является слишком обширной темой, чтобы обсуждать её в этом сборнике рецептов. Однако если вам нужно работать с матрицами и векторами, начните именно с NumPy. За информацией о библиотеке отправляйтесь на <http://www.numpy.org>.

3.11. Случайный выбор

Задача

Вы хотите выбрать случайные элементы из последовательности или сгенерировать случайные числа.

Решение

Модуль `random` содержит разнообразные функции для генерации случайных чисел и выбора случайных элементов. Например, чтобы выбрать случайный элемент последовательности используйте `random.choice()`:

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
>>> random.choice(values)
```

```
3
>>> random.choice(values)
1
>>> random.choice(values)
4
>>> random.choice(values)
6
>>>
```

Чтобы получить выборку из N элементов, используйте *random.sample()*. Каждый элемент выбирается один раз, так что если значения в полученной выборке повторяются, то это разные элементы оригинальной последовательности, имеющие одинаковое значение:

```
>>> random.sample(values, 2)
[6, 2]
>>> random.sample(values, 2)
[4, 3]
>>> random.sample(values, 3)
[4, 3, 1]
>>> random.sample(values, 3)
[5, 4, 1]
>>>
```

Если вы хотите перемешать элементы в последовательности, используйте *random.shuffle()*:

```
>>> random.shuffle(values)
>>> values
[2, 4, 6, 5, 3, 1]
>>> random.shuffle(values)
>>> values
[3, 5, 2, 1, 6, 4]
>>>
```

Чтобы сгенерировать случайные целые числа, используйте *random.randint()*:

```
>>> random.randint(0,10)
2
>>> random.randint(0,10)
5
>>> random.randint(0,10)
0
>>> random.randint(0,10)
7
>>> random.randint(0,10)
10
```

```
>>> random.randint(0,10)
3
>>>
```

Чтобы сгенерировать одинаковые по формату числа с плавающей точкой в диапазоне от 0 до 1, используйте *random.random()*:

```
>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
>>>
```

Чтобы получить целое число из N случайных битов, используйте *random.getrandbits()*:

```
>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
>>>
```

Обсуждение

Модуль *random* вычисляет случайные числа, используя алгоритм «вихрь Мерсенна» (Mersenne twister, MT). Это детерминированный алгоритм, но вы можете изменить начальную инициализацию с помощью функции *random.seed()*:

```
random.seed() # Инициализация на базе системного времени или os.urandom()
random.seed(12345) # Инициализация на базе заданного целого числа
random.seed(b'bytedata') # Инициализация на базе байтовых данных
```

В добавок к уже продемонстрированной функциональности, *random* включает функции для равномерного, гауссового и других распределений вероятности. Например, *random.uniform()* вычисляет равномерно распределенные числа, а *random.gauss()* — нормально распределенные. За описанием других поддерживаемых распределений обратитесь к документации.

Функции в *random* не должны быть использованы в криптографических программах. Если вам нужна такая функциональность, обратитесь к функциям из модуля *ssl*. Например, *ssl.RAND_bytes()* может быть

использована для генерации криптографически безопасных последовательностей случайных байтов.

3.12. Перевод дней в секунды и другие базовые методы конвертации времени

Задача

Вашей программе требуется производить простые преобразования времени, такие как выражение дней в секундах, часов в минутах и т.д.

Решение

Чтобы производить конвертирование и арифметические операции над различными единицами времени, используйте модуль *datetime*. Например, чтобы представить интервал времени, создайте экземпляр *timedelta*:

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

Если вам нужно представить определенные даты и определенное время, создайте экземпляры *datetime* и проводите над ними обычные арифметические операции. Например:

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
```

```
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

Стоит отметить, что *datetime* знает о существовании високосных годов.

Например:

```
>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>
```

Обсуждение

Для самых базовых операций над датой и временем модуля *datetime* достаточно. Если перед вами стоят более сложные задачи, такие как работа с временными зонами, нечеткими интервалами времени, подсчет дат выходных дней и так далее, посмотрите на модуль [dateutil](#).

Например, многие подобные вычисления над временем могут быть выполнены с помощью функции *dateutil.relativedelta()*. Одна важная возможность заключается в том, что она заполняет разрывы, которые возникают при работе с месяцами (и отличающимся количеством дней в них).

Например:

```
>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this function
>>>

>>> from dateutil.relativedelta import relativedelta
```

```
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
>>>

>>> # Время между двумя датами
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>
```

3.13. Определение даты последней пятницы

Задача

Вы хотите создать общее решение для поиска даты ближайшего прошедшего дня недели — например, последней прошедшей пятницы.

Решение

В модуле `datetime` есть полезные функции и классы, которые помогают проводить такого рода вычисления. Хорошее обобщенное решение этой задачи выглядит как-то так:

```
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday']

def get_previous_byday(dayname, start_date=None):
    if start_date is None:
        start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
```

```
days_ago = (7 + day_num - day_num_target) % 7
if days_ago == 0:
    days_ago = 7
target_date = start_date - timedelta(days=days_ago)
return target_date
```

Использование этой функции в строке интерпретатора выглядит так:

```
>>> datetime.today() # Опорная точка
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Предыдущая неделя, не сегодня
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

Необязательный параметр *start_date* может быть предоставлен с использованием другого экземпляра *datetime*. Например:

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

Обсуждение

Этот рецепт работает путем отображения стартовой и интересующей даты на номера их позиций в неделе (где понедельник — это 0). Далее используется модульная арифметика, с её помощью мы вычисляем, сколько дней назад была нужная дата. Далее нужная дата высчитывается от стартовой даты путем вычитания соответствующего экземпляра *timedelta*.

Если вы выполняете много подобных вычислений, рекомендуем установить пакет [python-dateutil](#). Например, вот так можно выполнить аналогичную работу с использованием функции *relativedelta()* из модуля *dateutil*:

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111
>>> # Следующая пятница
```

```
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>
>>> # Последняя пятница
>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
>>>
```

3.14. Поиск диапазона дат для текущего месяца

Задача

У вас есть код, которому необходимо пройти в цикле по каждой дате текущего месяца, и вам нужен эффективный способ поиска диапазонов дат.

Решение

Прохождение в цикле по датам не требует предварительного создания списка всех дат. Вы можете просто вычислить стартовую и конечную даты в диапазоне, а затем использовать объекты *datetime.timedelta*, инкрементируя дату.

Вот функция, которая принимает любой объект *datetime* и возвращает кортеж, содержащий первую дату месяца и начальную дату следующего месяца:

```
from datetime import datetime, date, timedelta
import calendar

def get_month_range(start_date=None):
    if start_date is None:
        start_date = date.today().replace(day=1)
    _, days_in_month = calendar.monthrange(start_date.year, start_date.month)
    end_date = start_date + timedelta(days=days_in_month)
    return (start_date, end_date)
```

Получив эти данные, очень просто пройти в цикле по диапазону дат:

```
>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
```

```
>>> while first_day < last_day:  
...     print(first_day)  
...     first_day += a_day  
...  
2012-08-01  
2012-08-02  
2012-08-03  
2012-08-04  
2012-08-05  
2012-08-06  
2012-08-07  
2012-08-08  
2012-08-09  
#... и так далее...
```

Обсуждение

Этот рецепт работает так: сначала вычисляется дата, соответствующая первому дню месяца. Быстрый способ сделать это — использовать метод *replace()* объектов *date* или *datetime*, чтобы присвоить атрибуту *days* значение 1. Приятно, что метод *replace()* создает объект того же типа, к которому он был применен. В данном случае, поскольку на входе у нас был экземпляр *date*, результат тоже является экземпляром *date*. Точно так же мы бы получили экземпляр *datetime*, если бы на входе у нас был экземпляр *datetime*.

Затем функция *calendar.monthrange()* используется для нахождения количества дней в рассматриваемом месяце. Модуль *calendar* весьма полезен для получения базовых данных о календарях. Функция *monthrange()* возвращает кортеж, который содержит день недели и количество дней в месяце.

Когда мы знаем количество дней в месяце, конечная дата вычисляется путём добавления соответствующего *timedelta* к стартовой дате. Тонкий, но важный аспект этого рецепта — конечная дата не включается в диапазон (на самом деле это первая дата следующего месяца). Это отражает присущее срезам и диапазонам Python поведение, которое также не подразумевает включение последнего элемента.

Чтобы пройти в цикле по диапазону дат, используются стандартные математические операции и операторы сравнения. Например, экземпляр *timedelta* может быть использован для инкрементирования даты. Оператор *<* используется для проверки того, не достигнута ли конечная дата.

В идеальном случае стоит создать функцию, которая будет работать как встроенная `range()`, но с датами. К счастью, есть чрезвычайно простой способ сделать это с помощью генератора:

```
def date_range(start, stop, step):
    while start < stop:
        yield start
        start += step
```

Вот пример её использования:

```
>>> for d in date_range(datetime(2012, 9, 1), datetime(2012, 10, 1),
timedelta(hours=6)):
...     print(d)
...
2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
2012-09-02 00:00:00
2012-09-02 06:00:00
...
>>>
```

Повторимся, самое большое преимущество такой реализации в том, что датами и временем можно манипулировать с помощью стандартных математических операторов и операторов сравнения.

3.15. Конвертирование строк в даты и время

Задача

Ваше приложение получает данные о времени в строковом формате, но вы хотите конвертировать их в объекты `datetime`, чтобы выполнять над ними нестроковые операции.

Решение

Стандартный модуль `datetime` обычно легко справляется с этой задачей. Например:

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)
>>>
```

Обсуждение

Метод `datetime.strptime()` поддерживает множество параметров форматирования, такие как `%Y` для года из четырёх цифр и `%m` для месяца из двух цифр. Также стоит отметить, что эти параметры-плейсхолдеры работают в обратном направлении, что поможет, если вам нужно вывести объект `datetime` в строке, и при этом заставить его красиво выглядеть.

Предположим, например, что у ваша программа генерирует объект `datetime`, но вам нужно создать из него красивую, понятную людям дату, чтобы потом вставить ее в заголовок автоматически создаваемого письма или отчёта:

```
>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
>>>
```

Стоит отметить, что производительность метода `strptime()` часто оказывается намного хуже, чем вы могли бы ожидать, поскольку функция написана на чистом Python и должна работать со всеми установками системной локализации. Если вы парсите множество дат в своей программе и знаете их точный формат, вы можете добиться намного более высокой производительности путём написания собственного решения. Например, если вы знаете, что даты представлены в формате “YYYY-MM-DD”, вы могли бы написать такую функцию:

```
from datetime import datetime
def parse_ymd(s):
    year_s, mon_s, day_s = s.split('-')
    return datetime(int(year_s), int(mon_s), int(day_s))
```

При тестировании эта функции оказалась более чем в 7 раз быстрее метода

`datetime.strptime()`. Это стоит держать в голове, если вы обрабатываете большие объемы данных с датами.

3.16. Манипулирование датами с учётом таймзон

Задача

У вас назначена телефонная конференция на 21 декабря 2012 года в 9:30 а.м. по чикагскому времени. В какое локальное время ваш друг из индийского города Бангалор должен выйти на связь?

Решение

Для практических любых задач, связанных с таймзонами, вы можете использовать модуль `pytz`. Этот пакет предоставляет базу таймзон Олсона (`tz database`), которая является стандартом де-факто для многих языков программирования и операционных систем.

Большая часть случаев использования `pytz` приходится на приведение к локальному времени дат, созданных с помощью библиотеки `datetime`. Например, вот как вы могли бы представить дату с чикагским местным временем:

```
>>> from datetime import datetime
>>> from pytz import timezone
>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Локализуем дату для Чикаго
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
>>>
```

Когда дата локализована (привязана к местному времени), ее можно конвертировать в другие таймзоны. Чтобы найти бангалорское время, вы можете сделать так:

```
>>> # Преобразуем во время по Бангалору
>>> bang_d = loc_d.astimezone(timezone('Asia/Kolkata'))
>>> print(bang_d)
2012-12-21 21:00:00+05:30
>>>
```

Если вы собираетесь выполнять арифметические операции над локализованными датами, вам нужно знать о переводах времени с летнего на зимнее и прочих подобных деталях. Например, в 2013 году стандартное летнее время США началось 13 марта в 2:00 ночи по местному времени городов (время было переведено на час вперед). Если бы провели стандартную арифметическую операцию над датами, то получили бы неверный результат. Например:

```
>>> d = datetime(2013, 3, 10, 1, 45)
>>> loc_d = central.localize(d)
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> later = loc_d + timedelta(minutes=30)
>>> print(later)
2013-03-10 02:15:00-06:00
# НЕВЕРНО! НЕВЕРНО!
>>>
```

Ответ получается неверным, поскольку он не учитывает перевод местного времени на один час. Чтобы исправить это, используйте метод таймзон `normalize()`. Например:

```
>>> from datetime import timedelta
>>> later = central.normalize(loc_d + timedelta(minutes=30))
>>> print(later)
2013-03-10 03:15:00-05:00
>>>
```

Обсуждение

Чтобы предотвратить взрыв головы, используйте обычную стратегию работы с локальным временем: преобразование всех дат в UTC и использование уже их для хранения и обработки. Например:

```
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> utc_d = loc_d.astimezone(pytz.utc)
```

```
>>> print(utc_d)
2013-03-10 07:45:00+00:00
>>>
```

Если время уже в UTC, вы можете не волноваться по поводу проблем, связанных с переходом на летнее время, а также прочих подобных вещах. Вы свободно можете выполнять арифметические операции с датами. Если же вы хотите вывести дату в локальном времени, просто сконвертируйте в нужную таймзону. Например:

```
>>> later_utc = utc_d + timedelta(minutes=30)
>>> print(later_utc.astimezone(central))
2013-03-10 03:15:00-05:00
>>>
```

С использованием таймзон есть одна проблема: какие имена таймзон использовать? Например, в этом рецепте мы как-то узнали, что “Asia/Kolkata” — это правильное название таймзоны для Индии. Чтобы узнать название нужной зоны, поищите в словаре `pytz.country_timezones`, указывая в качестве ключа код страны по ISO 3166. Например:

```
>>> pytz.country_timezones['IN']
['Asia/Kolkata']
>>>
```

К тому времени, как вы это прочтёте, модуль `pytz` может быть признан устаревшим, а ему на смену придёт улучшенная поддержка таймзон по [PEP 431](#). Однако многие из описанных проблем все равно нужно будет учитывать (вопросы работы с UTC и т.п.)

4. Итераторы и генераторы

Итерации — одна из сильнейших сторон Python. На высшем уровне абстракции вы можете рассматривать итерации как способ обработки элементов последовательности. Однако возможности намного шире: они включают создание собственных объектов-итераторов, применение полезных паттернов итераций из модуля `itertools`, создание функций-генераторов и т.д. Эта глава рассматривает типичные задачи, связанные с итерациями.

4.1. Ручное прохождение по итератору

Задача

Вам нужно обработать элементы итерируемого объекта, но по какой-то причине вы не хотите использовать цикл.

Решение

Чтобы вручную пройти по итерируемому объекту, используйте функцию `next()` и напишите код так, чтобы он ловил исключение `StopIteration`. Например, в этом случае мы вручную читаем строки из файла:

```
with open('/etc/passwd') as f:
    try:
        while True:
            line = next(f)
            print(line, end=' ')
    except StopIteration:
        pass
```

Обычно `StopIteration` используется для передачи сигнала о конце итерирования. Однако если вы используете `next()` вручную, вы вместо этого можете запрограммировать возвращение конечного значения, такого как `None`. Например:

```
with open('/etc/passwd') as f:
    while True:
        line = next(f, None)
        if line is None:
            break
        print(line, end=' ')
```

Обсуждение

В большинстве случаев для прохода по итерируемому объекту используется цикл `for`. Однако задачи иногда требуют более точного контроля лежащего в основе механизма итераций. Также это полезно для того, чтобы разобраться, как он работает.

Следующий интерактивный пример иллюстрирует базовые механизмы того,

что происходит во время итерирования:

```
>>> items = [1, 2, 3]
>>> # Получаем итератор
# Вызываем items.__iter__()
>>> it = iter(items)
>>> # Запускаем итератор
>>> next(it)
# Вызываем it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Последующие рецепты в этой главе раскрывают подробности о приёмах итерирования, что предполагает знание базового протокола итераторов. Убедитесь, что этот первый рецепт прочно улёгся у вас в памяти.

4.2. Делегирование итерации

Задача

Вы создали нестандартный объект-контейнер, который внутри содержит список, кортеж или какой-то другой итерируемый объект. Вы хотите заставить ваш новый контейнер работать с итерациями.

Решение

В типичном случае вам нужно определить метод `__iter__()`, который делегирует итерацию внутреннему содержимому контейнера. Например:

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)
```

```
def add_child(self, node):
    self._children.append(node)

def __iter__(self):
    return iter(self._children)

# Пример
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    for ch in root:
        print(ch)
    # Выводит Node(1), Node(2)
```

В этой программе метод `__iter__()` просто перенаправляет запрос на итерацию содержащемуся внутри атрибуту `_children`.

Обсуждение

Протокол итераций Python требует, чтобы `__iter__()` возвращал специальный объект-итератор, в котором реализован метод `__next__()`, который и выполняет итерацию. Если вы просто итерируете по содержимому другого контейнера, вам не стоит беспокоиться о деталях внутреннего механизма процесса. Вам нужно просто передать запрос на итерацию.

Использование функции `iter()` здесь позволяет «срезать путь» и написать более чистый код. `iter(s)` просто возвращает внутренний итератор, вызывая `s.__iter__()` — примерно так же, как `len(s)` вызывает `s.__len__()`.

4.3. Создание новых итерационных паттернов с помощью генераторов

Задача

Вы хотите реализовать собственный паттерн итераций, который будет отличаться от обычных встроенных функций (таких как `range()`, `reversed()` и т.п.)

Решение

Если вы хотите реализовать новый тип итерационного паттерна, определите его с помощью генератора. Вот, например, генератор, который создает диапазон чисел с плавающей точкой:

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

Чтобы использовать такую функцию, вы должны проитерировать по ней в цикле или применить ее с какой-то другой функцией, которая потребляет итерируемый объект (например, `sum()`, `list()` и т.п.) Например:

```
>>> for n in frange(0, 4, 0.5):
...     print(n)
...
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

Обсуждение

Само присутствие инструкции `yield` в функции превращает её в генератор. В отличие от обычной функции, генератор запускается только в ответ на итерацию. Вот эксперимент, который вы можете провести, чтобы понять внутренний механизм работы таких функций:

```
>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...         yield n
...         n -= 1
...     print('Done!')
```

```
...  
  
>>> # Создаёт генератор – обратите внимание на отсутствие вывода  
>>> c = countdown(3)  
>>> c  
<generator object countdown at 0x1006a0af0>  
  
>>> # Выполняется до первого yield и выдаёт значение  
>>> next(c)  
Starting to count from 3  
3  
  
>>> # Выполняется до следующего yield  
>>> next(c)  
2  
  
>>> # Выполняется до следующего yield  
>>> next(c)  
1  
  
>>> # Выполняется до следующего yield (итерирование останавливается)  
>>> next(c)  
Done!  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
StopIteration  
>>>
```

Ключевая особенность функции-генератора состоит в том, что она запускается только в ответ на операции `next` в ходе итерирования. Когда генератор возвращает значение, итерирование останавливается. Однако цикл `for`, который обычно используется для выполнения итераций, сам заботится об этих деталях, поэтому в большинстве случаев вам не стоит волноваться о них.

4.4. Реализация протокола итератора

Задача

Вы создаете собственные объекты, которые вы хотите сделать итерируемыми, и ищете простой способ реализовать протокол итератора.

Решение

На текущий момент простейший способ реализации итерируемости в объекте — это использование генератора. В [рецепте 4.2.](#) был представлен класс *Node*, представляющий древовидные структуры. Возможно, вы захотите реализовать итератор, который будет обходить узлы поиском в глубину. Вот как можно это сделать:

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        yield self
        for c in self:
            yield from c.depth_first()

# Пример
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
# Выводит Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)
```

В этой программе метод *depth_first()* просто прочесть и описать. Сначала он выдает себя, а затем итерируется по каждому потомку, выдавая элементы, производимые методом *depth_first()* потомка (используя *yield from*).

Обсуждение

Протокол итератора Python требует *__iter__()*, чтобы вернуть специальный

объект итератора, в котором реализована операция `_next()`, а исключение `StopIteration` используется для подачи сигнала о завершении. Однако создание таких объектов часто может стать запутанным делом. Например, следующая программа демонстрирует альтернативную реализацию метода `depth_first()`, использующую ассоциированный класс итератора:

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__():
        return 'Node({!r})'.format(self._value)

    def add_child(self, other_node):
        self._children.append(other_node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)

class DepthFirstIterator(object):
    ...
    Depth-first traversal
    ...

    def __init__(self, start_node):
        self._node = start_node
        self._children_iter = None
        self._child_iter = None

    def __iter__(self):
        return self

    def __next__(self):
        # Возвращает себя, если только что запущен, создаёт итератор для потомков
        if self._children_iter is None:
            self._children_iter = iter(self._node)
            return self._node

        # Если обрабатывает потомка, возвращает его следующий элемент
        elif self._child_iter:
            try:
                nextchild = next(self._child_iter)
                return nextchild
            except StopIteration:
                self._child_iter = None

        # Иначе возвращаем текущего потомка
        nextchild = next(self._children_iter)
        self._child_iter = iter(nextchild)
        return nextchild
```

```
        return next(self)

    # Переходим к следующему потомку и начинаем итерировать по нему
    else:
        self._child_iter = next(self._children_iter).depth_first()
        return next(self)
```

Класс DepthFirstIterator работает так же, как и версия на основе генератора, но он беспорядочен и некрасив, поскольку итератор вынужден хранить много сложных состояний о состоянии итерационного процесса. Откровенно говоря, никому не нравится писать такой мозговыносящий код. Реализуйте итератор на базе генератора и успокойтесь на этом.

4.5. Итерирование в обратном порядке

Задача

Вы хотите проитерировать по последовательности в обратном порядке.

Решение

Используйте встроенную функцию `reversed()`. Например:

```
>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...
4
3
2
1
```

Обратная итерация сработает только в том случае, если объект имеет определенный размер, или если в нём реализован специальный метод `__reversed__()`. Если ни одно из этих условий не выполнено, вы должны будете сначала конвертировать объект в список. Например:

```
# Выводит файл задом наперёд
f = open('somefile')
for line in reversed(list(f)):
    print(line, end='')
```

Обратите внимание, что конвертирование итерируемого объекта в список может съесть много памяти, если список получится большим.

Обсуждение

Многие программисты не знают, что итерирование в обратном порядке может быть переопределено в собственном классе, если он реализует метод `__reversed__()`. Например:

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # Прямой итератор
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    # Обратный итератор
    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1
```

Определение обратного итератора делает код намного более эффективным, а также снимает необходимость предварительного помещения данных в список для выполнения итераций в обратном порядке.

4.6. Определение генератора с дополнительным состоянием

Задача

Вы хотите написать генератор, но функция работает с дополнительным состоянием, которое вам хотелось бы каким-то образом показать пользователю.

Решение

Если вам нужен генератор, который показывает пользователю дополнительное состояние, не забудьте, что вы можете легко реализовать его в форме класса, поместив код генератора в метод `__iter__()`. Например:

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines, 1):
            self.history.append((lineno, line))
            yield line

    def clear(self):
        self.history.clear()
```

Вы можете обращаться с этим классом так же, как с обычным генератором. Однако, поскольку он создает экземпляр, вы можете обращаться к внутренним атрибутам, таким как `history` или метод `clear()`. Например:

```
with open('somefile.txt') as f:
    lines = linehistory(f)
    for line in lines:
        if 'python' in line:
            for lineno, hline in lines.history:
                print('{}:{}'.format(lineno, hline), end=' ')
```

Обсуждение

С генераторами легко попасть в ловушку, если пытаться делать всё только с помощью функций. В результате может получиться сложный код, если генератору нужно взаимодействовать с другими частями программы некими необычными способами (раскрытие атрибутов, разрешение на управление через вызов методов и т.п.) В этом случае просто используйте определение класса, как показано выше. Определение генератора в методе `__iter__()` не изменит ничего в том, как вы напишете алгоритм. Но тот факт, что генератор станет частью класса, упростит задачу предоставления пользователям атрибутов и методов для каких-то взаимодействий.

Потенциальная хрупкость показанного приёма заключается в том, что он

может потребовать дополнительного шага: вызова `iter()`, если вы собираетесь провести итерацию не через цикл `for`. Например:

```
>>> f = open('somefile.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator

>>> # Сначала вызываем iter(), затем начинаем итерирование
>>> it = iter(lines)
>>> next(it)
'hello world\n'
>>> next(it)
'this is a test\n'
>>>
```

4.7. Получение среза итератора

Задача

Вы хотите получить срез данных, производимых итератором, но обычный оператор среза не работает.

Решение

Функция `itertools.islice()` отлично подходит для получения срезов генераторов и итераторов. Например:

```
>>> def count(n):
...     while True:
...         yield n
...         n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable

>>> # Теперь используем islice()
>>> import itertools
>>> for x in itertools.islice(c, 10, 20):
```

```
...     print(x)
...
10
11
12
13
14
15
16
17
18
19
>>>
```

Обсуждение

Из итераторов и генераторов получить срез напрямую нельзя, потому что отсутствует информация об их длине (и в них не реализовано индексирование). Результат *islice()* — это итератор, который создает элементы нужного среза, но делает это путем потребления и выбрасывания всех элементов до стартового индекса среза. Следующие элементы затем производятся объектом *islice*, пока не будет достигнут конечный индекс среза.

Важно отметить, что *islice()* будут потреблять данные, предоставляемые итератором. Это важно, поскольку итераторы не могут быть отмотаны назад. Если вам нужно возвращаться назад, то вам, наверное, лучше сначала конвертировать данные в список.

4.8. Пропуск первой части итерируемого объекта

Задача

Вы хотите итерировать по элементам в последовательности, но первые несколько элементов вам неинтересны, и вы хотите их опустить.

Решение

В модуле *itertools* есть несколько функций, которые могут быть использованы для решения этой задачи. Первая — *itertools.dropwhile()*. Чтобы использовать

её, вы предоставляете функцию и итерируемый объект. Возвращаемый итератор отбрасывает первые элементы в последовательности до тех пор, пока предоставленная функция возвращает True. А затем выдаётся вся оставшаяся последовательность.

Предположим, что вы читаете файл, который начинается со строчек с комментариями:

```
>>> with open('/etc/passwd') as f:  
...     for line in f:  
...         print(line, end=' ')  
...  
##  
# User Database  
#  
# Note that this file is consulted directly only when the system is run  
# in single-user mode. At other times, this information is provided by  
# Open Directory.  
...  
##  
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false  
root:*:0:0:System Administrator:/var/root:/bin/sh  
...  
>>>
```

Если вы хотите пропустить все начальные закомментированные строчки, вот как это можно сделать:

```
>>> from itertools import dropwhile  
>>> with open('/etc/passwd') as f:  
...     for line in dropwhile(lambda line: line.startswith('#'), f):  
...         print(line, end=' ')  
...  
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false  
root:*:0:0:System Administrator:/var/root:/bin/sh  
...  
>>>
```

Этот пример показывает, как можно пропустить первые элементы в соответствии с возвращаемым значением проверочной функции. Если так случилось, что вы знаете точное количество элементов, которые хотите пропустить, то вы можете вместо вышеописанного способа использовать `itertools.islice()`. Например:

```
>>> from itertools import islice
```

```
>>> items = [ 'a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...     print(x)
...
1
4
10
15
>>>
```

В этом примере последний аргумент `islice()` `None` необходим для того, чтобы обозначить, что вам нужно всё за пределами первых трёх элементов (а не первые три элемента). То есть срез `[3:0]`, а не `[:3]`.

Обсуждение

Главное преимущество функций `dropwhile()` и `islice()` в том, что они позволяют избежать написания грязного кода наподобие вот такого:

```
with open('/etc/passwd') as f:
    # Пропускаем начальные комментарии
    while True:
        line = next(f, '')
        if not line.startswith('#'):
            break

    # Обрабатываем оставшиеся строки
    while line:
        # Можно заменить полезной обработкой
        print(line, end=' ')
        line = next(f, None)
```

Отбрасывание первой части итерируемого объекта также немного отличается от простого фильтрования. Например, первая часть этого рецепта может быть переписана вот так:

```
with open('/etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))
    for line in lines:
        print(line, end=' ')
```

Очевидно, что это отбросит все закомментированные строчки в начале файла, но такое решение отбросит и все остальные такие строчки во всём файле. С другой стороны, решение, которое отбрасывает все элементы до тех пор, пока не будет встречен элемент, не соответствующий условиям

отбрасывания, подходит под наши требования: все последующие элементы будут возвращены без фильтрования.

Стоит отметить, что этот рецепт работает со всеми итерируемыми объектами, включая те, размер которых нельзя оценить заранее: генераторами, файлами и другими подобными объектами.

4.9. Итерирование по всем возможным комбинациям и перестановкам

Задача

Вы хотите проитерировать по всем возможным комбинациям и перестановкам коллекции элементов.

Решение

Модуль *itertools* предоставляет три функции, подходящие для этой задачи. Первая, *itertools.permutations()*, принимает коллекцию элементов и создает последовательность кортежей со всеми возможными перестановками (то есть она тасует их во всех возможных конфигурациях). Например:

```
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
...     print(p)
...
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

Если вы хотите получить все возможные перестановки меньшей длины, вы можете передать функции необязательный аргумент со значением длины. Например:

```
>>> for p in permutations(items, 2):
...     print(p)
```

```
...
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>
```

Используйте `itertools.combinations()`, чтобы создать последовательность комбинаций элементов входной последовательности. Например:

```
>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')
>>> for c in combinations(items, 2):
...     print(c)
...
('a', 'b')
('a', 'c')
('b', 'c')
>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>
```

Для функции `combinations()` порядок элементов не имеет значения. Комбинацию ('a', 'b') она считает аналогичной ('b', 'a') — поэтому вторая в выводимых результатах отсутствует.

При создании комбинаций выбранные элементы удаляются из коллекции возможных кандидатов (то есть если 'a' уже выбран, он больше не будет рассматриваться). А функция `itertools.combinations_with_replacement()` выбирает один и тот же элемент более одного раза. Например:

```
>>> for c in combinations_with_replacement(items, 3):
...     print(c)
...
('a', 'a', 'a')
('a', 'a', 'b')
('a', 'a', 'c')
('a', 'b', 'b')
```

```
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'b', 'b')
('b', 'b', 'c')
('b', 'c', 'c')
('c', 'c', 'c')
>>>
```

Обсуждение

Этот рецепт показывает лишь небольшую часть мощи модуля *itertools*. Хотя вы могли бы самостоятельно написать код, который выполняет перестановки и комбинации, это, вероятно, отняло бы у вас больше пары секунд времени. Когда вы сталкиваетесь с нетривиальными задачами в сфере итераций, обратитесь к *itertools*, это всегда окупается. Если задача распространённая, велик шанс того, что вы найдете готовое решение.

4.10. Итерирование по парам «индекс-значение» последовательности

Задача

Вы хотите проитерировать по последовательности и при этом хранить информацию о том, какой по счёту элемент сейчас обрабатывается.

Решение

Встроенная функция `enumerate()` изящно справляется с этой задачей:

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list):
...     print(idx, val)
...
0 a
1 b
2 c
```

Для печати вывода с привычными номерами строк (то есть с нумерацией, начинающейся с 1, а не с 0), вы можете передать соответствующий аргумент `start`:

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list, 1):
...     print(idx, val)
...
1 a
2 b
3 c
```

Этот приём особенно полезен для учёта номеров строк в файлах, если нужно будет вывести номер строки в сообщении об ошибке:

```
def parse_data(filename):
    with open(filename, 'rt') as f:
        for lineno, line in enumerate(f, 1):
            fields = line.split()
            try:
                count = int(fields[1])
            ...
            except ValueError as e:
                print('Line {}: Parse error: {}'.format(lineno, e))
```

`enumerate()` удобна, например, для отслеживания смещения (`offset`) в списке для вхождений определенных значений. Так что если вы хотите отобразить слова в файле к строчкам, в которых они встречаются, это легко сделать с помощью `enumerate()` — функция отображает каждое слово на смещение строки в файле, где оно найдено:

```
word_summary = defaultdict(list)

with open('myfile.txt', 'r') as f:
    lines = f.readlines()

for idx, line in enumerate(lines):
    # Создаёт список слов в текущей строке
    words = [w.strip().lower() for w in line.split()]
    for word in words:
        word_summary[word].append(idx)
```

Если вы выведете `word_summary` после обработки файла, это будет словарь (`default dict`, если быть точными), и каждое слово будет ключом. Значение каждого ключа — список номеров строк, где встретилось это слово. Если слово встретилось дважды в одной строке, этот номер строки будет записан в список дважды, что делает возможным получение разнообразных простых метрик текста.

Обсуждение

`enumerate()` — симпатичное решение для ситуаций, где вы могли бы склоняться к использованию собственной переменной-счетчика. Вы могли бы написать такой код:

```
lineno = 1
for line in f:
    # Обработка строки
    ...
lineno += 1
```

Но часто более элегантным (и менее подверженным ошибкам) способом становится использование `enumerate()`:

```
for lineno, line in enumerate(f):
    # Обработка строки
    ...
```

Значение, возвращаемое функцией `enumerate()`, является объектом `enumerate`. Это итератор, который последовательно возвращает кортежи, состоящие из счётчика и значения, возвращаемого вызовом функции `next()` для последовательности, которую вы обходите.

Стоит отметить, что иногда можно запутаться при применении `enumerate()` к последовательности кортежей, которые при этом распаковываются:

```
data = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

# Верно!
for n, (x, y) in enumerate(data):
    ...

# Ошибка!
for n, x, y in enumerate(data):
    ...
```

4.11. Одновременное итерирование по нескольким последовательностям

Задача

Вы хотите за один раз проитерировать по элементам, содержащимся более чем в одной последовательности.

Решение

Чтобы итерировать по более чем одной последовательности за раз, используйте функцию `zip()`. Например:

```
>>> xpts = [1, 5, 4, 2, 10, 7]
>>> ypts = [101, 78, 37, 15, 62, 99]
>>> for x, y in zip(xpts, ypts):
...     print(x,y)
...
1 101
5 78
4 37
2 15
10 62
7 99
>>>
```

`zip(a, b)` работает путём создания итератора, который производит кортежи (x, y) , где x берётся из a , а y – из b . Итерирование останавливается, когда заканчивается одна из последовательностей. Поэтому результат будет таким же по длине, как и самая короткая из входных последовательностей.

Например:

```
>>> a = [1, 2, 3]
>>> b = ['w', 'x', 'y', 'z']
>>> for i in zip(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
```

Если такое поведение нежелательно, используйте функцию `itertools.zip_longest()`. Например:

```
>>> from itertools import zip_longest
>>> for i in zip_longest(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
```

```
(3, 'y')
(None, 'z')
>>> for i in zip_longest(a, b, fillvalue=0):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(0, 'z')
>>>
```

Обсуждение

`zip()` обычно используется тогда, когда вам нужно создать пары из данных. Предположим, например, что у вас есть список заголовков столбцов и значения столбцов:

```
headers = ['name', 'shares', 'price']
values = ['ACME', 100, 490.1]
```

Используя `zip()`, вы можете создать пары значений и поместить их в словарь:

```
s = dict(zip(headers, values))
```

Если вы хотите вывести результат, можно поступить так:

```
for name, val in zip(headers, values):
    print(name, '=', val)
```

Менее распространённое применение `zip()` заключается в том, что функции может быть передано не две последовательности, а больше. В этом случае кортежи результата будут иметь такое количество элементов, каким было количество последовательностей. Например:

```
>>> a = [1, 2, 3]
>>> b = [10, 11, 12]
>>> c = ['x', 'y', 'z']
>>> for i in zip(a, b, c):
...     print(i)
...
(1, 10, 'x')
(2, 11, 'y')
(3, 12, 'z')
>>>
```

И последнее: важно подчеркнуть, что `zip()` возвращает итератор. Если вам нужны сохраненные в списке спаренные значения, используйте функцию `list()`. Например:

```
>>> zip(a, b)
<zip object at 0x1007001b8>
>>> list(zip(a, b))
[(1, 10), (2, 11), (3, 12)]
>>>
```

4.12. Итерирование по элементам, находящимся в отдельных контейнерах

Проблема

Вам нужно выполнить одинаковую операцию над большим количеством объектов, но объекты находятся в различных контейнерах, а вам хотелось бы избежать написания вложенных циклов, причем без потери читабельности кода.

Решение

Для упрощения этой задачи можно использовать метод `itertools.chain()`. Он принимает список итерируемых объектов и возвращает итератор, который скрывает тот факт, что вы на самом деле работаете с несколькими контейнерами. Рассмотрим пример:

```
>>> from itertools import chain
>>> a = [1, 2, 3, 4]
>>> b = ['x', 'y', 'z']
>>> for x in chain(a, b):
...     print(x)
...
1
2
3
4
x
y
z
>>>
```

Обычно `chain()` используется, если вы хотите выполнить некоторые операции над всеми элементами за один раз, но элементы разнесены по разным рабочим наборам. Например:

```
# Различные наборы элементов
active_items = set()
inactive_items = set()

# Итерируем по всем элементам
for item in chain(active_items, inactive_items):
    # Обработка элемента
    ...

```

Это решение намного более элегантно, нежели использование двух отдельных циклов, как показано в этом примере:

```
for item in active_items:
    # Обработка элемента
    ...

for item in inactive_items:
    # Обработка элемента
    ...

```

Обсуждение

`itertools.chain()` принимает один или более итерируемых объектов в качестве аргументов. Далее она создает итератор, который последовательно потребляет и возвращает элементы, производимые каждым из предоставленных итерируемых объектов. Это тонкое различие, но `chain()` эффективнее, чем итерирование по предварительно объединённым последовательностям. Например:

```
# Неэффективно
for x in a + b:
    ...

# Уже лучше
for x in chain(a, b):
    ...

```

В первом случае операция `a + b` создает новую последовательность и дополнительно требует, чтобы `a` и `b` относились к одному типу. `chain()` не

выполняет такую операцию, намного эффективнее обращается с памятью, если входные последовательности большие, а также легко применяется к итерируемым объектам различных типов.

4.13. Создание каналов для обработки данных

Задача

Вы хотите обрабатывать данные итеративно, в стиле обрабатывающего данные канала (похожего на канал Unix — он же конвейер). Например, у вас есть огромный объем данных для обработки, который просто не поместится в память целиком.

Решение

Генераторы хорошо подходят для реализации обрабатывающих каналов. Предположим, например, что у вас есть огромный каталог с файлами логов, который вы хотите обработать:

```
foo/
    access-log-012007.gz
    access-log-022007.gz
    access-log-032007.gz
    ...
    access-log-012008
bar/
    access-log-092007.bz2
    ...
    access-log-022008
```

Предположим, каждый файл содержит такие строки данных:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 1187
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 4
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 1
...
```

Чтобы обработать эти файлы, вы могли бы создать коллекцию небольших

генераторов, которые будут выполнять специфические замкнутые в себе задачи:

```
import os
import fnmatch
import gzip
import bz2
import re

def gen_find(filepat, top):
    ...
    Находит все имена файлов в дереве каталогов,
    которые совпадают с шаблоном маски оболочки
    ...
    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist, filepat):
            yield os.path.join(path, name)

def gen_opener(filenames):
    ...
    Открывает последовательность имён файлов, которая
    раз за разом производит файловый объект.
    Файл закрывается сразу же после перехода
    к следующему шагу итерации.
    ...
    for filename in filenames:
        if filename.endswith('.gz'):
            f = gzip.open(filename, 'rt')
        elif filename.endswith('.bz2'):
            f = bz2.open(filename, 'rt')
        else:
            f = open(filename, 'rt')
        yield f
        f.close()

def gen_concatenate(iterators):
    ...
    Объединяет цепочкой последовательность
    итераторов в одну последовательность.
    ...
    for it in iterators:
        yield from it

def gen_grep(pattern, lines):
    ...
    Ищёт шаблон регулярного выражения
    в последовательности строк
    ...
    pat = re.compile(pattern)
```

```
for line in lines:
    if pat.search(line):
        yield line
```

Теперь вы можете легко совместить эти функции для создания обрабатывающего канала. Например, чтобы найти все файлы логов, которые содержат слово *python*, вы можете поступить так:

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
for line in pylines:
    print(line)
```

Если вы хотите ещё расширить функциональность канала, вы можете скармливать данные выражениям-генераторам. Например, эта версия находит количество переданных байтов и подсчитывает общую сумму:

```
lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
bytecolumn = (line.rsplit(None, 1)[1] for line in pylines)
bytes = (int(x) for x in bytecolumn if x != '-')
print('Total', sum(bytes))
```

Обсуждение

Обработка данных в «каналообразной» манере отлично работает для решения широкого спектра задач: парсинга, чтения из реалтаймовых источников данных, периодического опрашивания и т.д.

Для понимания представленного выше кода главное уловить, что инструкция *yield* действует как своего рода производитель данных для цикла *for*, который действует как потребитель данных. Когда генераторы соединены, каждый *yield* скармливает один элемент данных следующему этапу канала, который потребляет его, совершая итерацию. В последнем примере функция *sum()* управляет всей программой, вытягивая один элемент за другим из канала (конвейера) генераторов.

Приятная особенность этого подхода заключается в том, что каждый генератор является маленьким и замкнутым на себе, поэтому их легко писать

и поддерживать. Во многих случаях они получаются настолько универсальными, что могут быть переиспользованы в других контекстах. Получающийся код, который «склеивает» компоненты вместе, тоже обычно читается как простой для понимания рецепт.

Есть небольшая тонкость с использованием функции `gen_concatenate()`. Ее назначение — конкатенировать входные последовательности в одну длинную последовательность строк. `itertools.chain()` выполняет похожую функцию, но требует, чтобы все объединяемые итерируемые объекты были определены в качестве аргументов. В случае этого конкретного рецепта, такой подход потребовал бы инструкции типа `lines = itertools.chain(*files)`, которая заставила бы генератор `gen_opener()` быть полностью потребленным. Поскольку генератор производит последовательность открытых файлов, которые немедленно закрываются на следующем шаге итерации, `chain()` использовать нельзя. Показанное решение позволяет решить эту проблему.

Также в функции `gen_concatenate()` используется `yield from` для делегирования субгенератору. Объявление `yield from it` просто заставляет `gen_concatenate()` выдать все значения, произведенные генератором `it`. Это описано далее, в **рецепте 4.14.**

И последнее: стоит отметить, что «конвейерный» («канальный») подход не сработает для всех на свете задач обработки данных. Иногда вам просто необходимо работать со всеми данными сразу. Однако даже в этом случае использование каналов генераторов может стать путём логического разбиения задачи.

Дэвид Бизли подробно написал об этих приёмах в обучающей презентации [«Трюки с генераторами для системных программистов»](#). Если вам нужны дополнительные примеры, обратитесь к ней.

4.14. Превращение вложенной последовательности в плоскую

Задача

У вас есть вложенная последовательность, и вы хотите превратить ее в один плоский список значений.

Решение

Это легко решается с помощью рекурсивного генератора с инструкцией *yield from*. Например:

```
from collections import Iterable

def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)
        else:
            yield x

items = [1, 2, [3, 4, [5, 6], 7], 8]

# Производит 1 2 3 4 5 6 7 8
for x in flatten(items):
    print(x)
```

В этой программе *isinstance(x, Iterable)* просто проверяет, является ли элемент итерируемым объектом. Если это так, то *yield from* используется в качестве некой подпрограммы, чтобы выдать все его значения. Конечный результат — одна последовательность без вложенности.

Дополнительный аргументы *ignore_types* и проверка *not isinstance(x, ignore_types)* нужны для предотвращения определения строк и байтов как итерируемых последовательностей, без чего они были бы разбиты на отдельные символы. Это позволяет вложенным спискам строк работать так, как большинство людей этого и ожидают:

```
>>> items = ['Dave', 'Paula', ['Thomas', 'Lewis']]
>>> for x in flatten(items):
...     print(x)
...
Dave
Paula
Thomas
Lewis
>>>
```

Обсуждение

Инструкция *yield from* — отличный способ написания генераторов, которые

вызывают другие генераторы в качестве подпроцедур. Без использования этой инструкции вам придётся вставить в код дополнительный цикл. Например:

```
def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            for i in flatten(x):
                yield i
        else:
            yield x
```

Хотя это незначительное изменение, инструкция *yield from* просто приятнее и делает код чище.

Как было отмечено, дополнительная проверка на строки и байты нужна для предотвращения их разбивки на отдельные символы. Если есть ещё какие-то типы, которые вы не хотите раскрывать, вы просто можете передать другие значения в *ignore_types*.

Стоит отметить, что *yield from* играет более важную роль в продвинутых программах, использующих корутины (сопрограммы) и основанную на генераторах многопоточность. См. другой пример в [рецепте 12.12](#).

4.15. Последовательное итерирование по слитым отсортированным итерируемым объектам

Задача

У вас есть коллекция отсортированных последовательностей, и вы хотите проитерировать по отсортированной последовательности этих последовательностей, слитых воедино.

Решение

Функция *heapq.merge()* делает именно это:

```
>>> import heapq
>>> a = [1, 4, 7, 10]
```

```
>>> b = [2, 5, 6, 11]
>>> for c in heapq.merge(a, b):
...     print(c)
...
1
2
4
5
6
7
10
11
```

Обсуждение

Итеративная природа `heapq.merge()` подразумевает, что она никогда не читает ни одну из переданных ей последовательностей сразу до конца. Это значит, что вы можете использовать ее на длинных последовательностях с очень незначительным оверхедом. Вот, например, как вы можете слить воедино два отсортированных файла:

```
import heapq
with open('sorted_file_1', 'rt') as file1, \
    open('sorted_file_2') 'rt' as file2, \
    open('merged_file', 'wt') as outf:

    for line in heapq.merge(file1, file2):
        outf.write(line)
```

Важно отметить, что `heapq.merge()` требует, чтобы все передаваемые ей последовательности уже были отсортированы. Она не читает предварительно данные в кучу, не выполняет предварительную сортировку. Также она не выполняет никакой валидации входных данных на соответствие требованиям упорядоченности. Она просто проверяет набор элементов из «голов» каждой переданной последовательности и выдает минимальный из найденных. Далее читается новый элемент из выбранной последовательности, и процесс повторяется до тех пор, пока все входные последовательности не будут полностью потреблены.

4.16. Замена бесконечных циклов `while` итератором

Задача

У вас есть код, который использует цикл `while` для итеративной обработки данных, потому что в программе присутствует функция или какое-то необычное проверочное условие, которое нельзя вместить в стандартный итерационный паттерн.

Решение

Вполне обычный код для программ, работающих с вводом-выводом:

```
CHUNKSIZE = 8192

def reader(s):
    while True:
        data = s.recv(CHUNKSIZE)
        if data == b'':
            break
        process_data(data)
```

Такой код часто можно заменить использованием `iter()`, как показано ниже:

```
def reader(s):
    for chunk in iter(lambda: s.recv(CHUNKSIZE), b''):
        process_data(data)
```

Если вы сомневаетесь, будет ли это работать, вы можете попробовать похожий пример для обработки файлов:

```
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''):
...     n = sys.stdout.write(chunk)
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
...
>>>
```

Обсуждение

Малоизвестная возможность встроенной функции *iter()* заключается в том, что она может опционально принимать вызываемый (callable) аргумент и «стража» (завершающее значение). При таком использовании функция создает итератор, который снова и снова повторяет вызов предоставленного вызываемого объекта, пока он не вернет значение, равное «стражу».

Этот конкретный подход хорошо работает с некоторыми типами многократно вызываемых функций, таких как операции ввода-вывода. Например, если вы хотите читать данные кусочками (чанками) из файлов или сокетов, вы обычно должны многократно вызывать *read()* или *recv()* с последующей проверкой достижения конца файла. Представленный выше рецепт просто берет эти две функциональности и совмещает в единственном вызове *iter()*. Использование *lambda* в решении необходимо для создания вызываемого объекта, который не принимает аргументов, но при этом поставляет аргумент нужного размера в *recv()* или *read()*.

5. Файлы и ввод-вывод

Всем программам нужно производить ввод и вывод. Эта глава покрывает типичные идиомы для работы с различными типами файлов, включая текстовые и бинарные, кодировки файлов и прочие связанные темы. Также тут освещены приёмы манипулирования именами файлов и каталогов.

5.1. Чтение и запись текстовых данных

Решение

Вам нужно прочитать или записать текстовые данные, представленные, возможно, в различных кодировках, таких как ASCII, UTF-8 или UTF-16.

Решение

Используйте функцию *open()* в режиме *rt* для чтения текстового файла. Например:

```
# Прочесть остаток файла в одну строку
with open('somefile.txt', 'rt') as f:
    data = f.read()
```

```
# Итерируем по строкам файла
with open('somefile.txt', 'rt') as f:
    for line in f:
        # process line
    ...
```

Похожим образом для записи в текстовый файл используйте `open()` в режиме `wt` (стирает и перезаписывает любое предыдущее содержание файла, если оно было):

```
# Пишем чанки (кусочки) текстовых данных
with open('somefile.txt', 'wt') as f:
    f.write(text1)
    f.write(text2)
    ...

# Перенаправленная инструкция print
with open('somefile.txt', 'wt') as f:
    print(line1, file=f)
    print(line2, file=f)
    ...
```

Чтобы добавить записываемый текст к концу существующего файла, используйте `open()` в режиме `at`.

По умолчанию файлы читаются и записываются в дефолтной системной кодировке, информацию о которой можно получить из `sys.getdefaultencoding()`. На большинстве компьютеров это будет `utf-8`. Если вы знаете, что текст, который вы читаете или пишете, представлен в другой кодировке, передайте необязательный параметр `encoding` функции `open()`. Например:

```
with open('somefile.txt', 'rt', encoding='latin-1') as f:
    ...
```

Python понимает несколько сотен текстовых кодировок. Однако самые распространенные — `ascii`, `latin-1`, `utf-8` и `utf-16`. `utf-8` обычно является безопасным выбором для работы с веб-приложениями. `ascii` соответствует 7-битным символам в диапазоне от U+0000 до U+007F. `latin-1` — это прямое отображение байтов 0-255 на символы Unicode от U-0000 до U-00FF. `latin-1` известна тем, что она никогда не вызовет ошибку декодирования при чтении текста в потенциально неизвестной кодировке. Чтение файла как `latin-1` может не привести к получению полностью правильно декодированного

текста, но этого бывает достаточно для извлечения полезных данных. Также, если вы позже запишете данные обратно, первоначальные данные будут сохранены.

Обсуждение

Чтение и запись файлов в большинстве случаев совершенно бесхитростны. Однако есть и тонкости. Во-первых, использование инструкции `with` в примере устанавливает контекст, в котором будут использованы файлы. Когда поток управления покидает блок `with`, файл будет автоматически закрыт. Вы не обязаны использовать инструкцию `with`, но если вы её не применяете, то не забудьте закрыть файл:

```
f = open('somefile.txt', 'rt')
data = f.read()
f.close()
```

Ещё одна небольшая сложность касается распознавания новых строк, символов которых отличаются в Unix и Windows (`\n` и `\r\n`). По умолчанию Python работает в так называемом «универсальном режиме новых строк». В этом режиме все распространённые символы новой строки распознаются, и все они конвертируются в единственный `\n` при чтении. Похожим образом символ новой строки `\r` конвертируется в дефолтный системный символ при выводе. Если вы не хотите использовать такую трансляцию, передайте функции `open()` аргумент `newline=''`:

```
# Читаем с отключённой трансляцией новой строки
with open('somefile.txt', 'rt', newline='') as f:
    ...
```

Чтобы продемонстрировать разницу, покажем, что вы увидите на компьютере с Unix, если вы читаете содержание файла в Windows-кодировке, в котором присутствуют сырье данные `hello world!\r\n`:

```
>>> # Трансляция новой строки включена (по умолчанию)
>>> f = open('hello.txt', 'rt')
>>> f.read()
'hello world!\n'

>>> # Трансляция новой строки отключена
>>> g = open('hello.txt', 'rt', newline='')
>>> g.read()
```

```
'hello world!\r\n'  
>>>
```

Последняя проблема касается возможных ошибок кодировки в текстовых файлах. При чтении или записи текстового файла вы можете натолкнуться на ошибку кодирования или декодирования. Например:

```
>>> f = open('sample.txt', 'rt', encoding='ascii')  
>>> f.read()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "/usr/local/lib/python3.3/encodings/ascii.py", line 26, in decode  
    return codecs.ascii_decode(input, self.errors)[0]  
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position  
12: ordinal not in range(128)  
>>>
```

Если вы получили эту ошибку, это обычно означает, что вы не используете правильную кодировку для чтения файла. Вы должны внимательно прочитать спецификацию того, что вы пытаетесь прочесть, и удостовериться, что вы делаете это правильно (то есть не читаете данные как UTF-8 вместо Latin-1 и т.п.) Если ошибки кодирования все еще возникают, вы можете передать необязательный аргумент `errors` функции `open()`, чтобы обрабатывать ошибки. Вот несколько примеров типичных схем обработки ошибок:

```
>>> # Заменяем плохие символы символом замены Unicode U+ffffd  
>>> f = open('sample.txt', 'rt', encoding='ascii', errors='replace')  
>>> f.read()  
'Spicy Jalape?o!'  
>>> # Полностью игнорируем плохие символы  
>>> g = open('sample.txt', 'rt', encoding='ascii', errors='ignore')  
>>> g.read()  
'Spicy Jalapeo!'  
>>>
```

Если вы постоянно ловите блох с кодировками, аргументом `errors` функции `open()` и изобретаете хаки, вы, вероятно, зря усложняете себе жизнь. Первое правило работы с текстом: убедитесь, что вы используете правильную кодировку. А если сомневаетесь, какую выбрать, используйте системную установку по умолчанию (обычно это UTF-8).

5.2. Перенаправление вывода в файл

Задача

Вы хотите перенаправить в файл вывод функции `print()`.

Решение

Используйте `print()` с именованным аргументом `file`:

```
with open('somefile.txt', 'rt') as f:  
    print('Hello World!', file=f)
```

Обсуждение

Про вывод в файл добавить больше и нечего. Разве что убедитесь, что файл открыт в текстовом режиме. Выводить в бинарном режиме так нельзя.

5.3. Вывод с другим разделителем или символом конца строки

Задача

Вы хотите вывести данные с помощью `print()`, но вы также хотите поменять символ-разделитель или символ конца строки.

Решение

Используйте именнованные аргументы `sep` и `end` с функцией `print()`, чтобы изменить вывод так, как вам нужно. Например:

```
>>> print('ACME', 50, 91.5)  
ACME 50 91.5  
>>> print('ACME', 50, 91.5, sep=', ')  
ACME,50,91.5  
>>> print('ACME', 50, 91.5, sep=', ', end='!!\n')  
ACME,50,91.5!!  
>>>
```

Использование аргумента `end` также позволяет подавить добавление символа новой строки при выводе. Например:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> for i in range(5):
...     print(i, end=' ')
...
0 1 2 3 4 >>>
```

Обсуждение

Использование `print()` с разными разделителями элементов часто является самым простым способом вывести данные, когда вам нужно сделать это с другим разделителем элементов. Однако иногда вы можете увидеть, как программисты используют `str.join()` для выполнения этой же задачи:

```
>>> print(', '.join('ACME', '50', '91.5'))
ACME, 50, 91.5
>>>
```

Проблема `str.join()` в том, что он работает только со строками. Это значит, что часто необходимо выполнить различные акробатические трюки, чтобы заставить его работать. Например:

```
>>> row = ('ACME', 50, 91.5)
>>> print(', '.join(row))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected str instance, int found
>>> print(', '.join(str(x) for x in row))
ACME, 50, 91.5
>>>
```

Вместо этого вы могли бы просто написать так:

```
>>> print(*row, sep=', ')
ACME, 50, 91.5
>>>
```

5.4. Чтение и запись бинарных данных

Задача

Вам нужно прочесть или записать бинарные данные, такие как содержимое картинок, звуковых файлов и т.п.

Решение

Используйте функцию `open()` в режиме `rb` или `wb`, чтобы читать и записывать бинарные данные. Например:

```
# Прочесть весь файл как одну байтовую строку
with open('somefile.bin', 'rb') as f:
    data = f.read()

# Записать бинарные данные в файл
with open('somefile.bin', 'wb') as f:
    f.write(b'Hello World')
```

При чтении бинарных данных важно подчеркнуть, что все получаемые данные будут в форме байтовых, а не текстовых строк. Похожим образом, при записи вы должны предоставить данные в форме объектов, которые представляют данные в форме байтов (байтовые строки, объекты `bytearray` и т.д.)

Обсуждение

При чтении бинарных данных тонкие сематические различия между байтовыми и текстовыми строками могут привести к проблемам. Нужно помнить, что индексирование и итерирование возвращают целочисленное байтовое значение, а не байтовые строки. Например:

```
>>> # Текстовая строка
>>> t = 'Hello World'
>>> t[0]
'H'
>>> for c in t:
...     print(c)
...
H
e
```

```
1
1
o
...
>>> # Байтовая строка
>>> b = b'Hello World'
>>> b[0]
72
>>> for c in b:
...     print(c)
...
72
101
108
108
111
...
>>>
```

Если вам когда-либо потребуется прочесть текст из или записать в открытый в бинарном режиме файл, убедитесь, что не забыли декодировать или закодировать его. Например:

```
with open('somefile.bin', 'rb') as f:
    data = f.read(16)
    text = data.decode('utf-8')

with open('somefile.bin', 'wb') as f:
    text = 'Hello World'
    f.write(text.encode('utf-8'))
```

Менее известный аспект бинарного ввода-вывода заключается в том, что такие объекты, как массивы и структуры языка С, могут быть использованы для записи без какого-либо промежуточного преобразования в объект *bytes*. Например:

```
import array
nums = array.array('i', [1, 2, 3, 4])
with open('data.bin', 'wb') as f:
    f.write(nums)
```

Это применимо к любому объекту, в котором реализован так называемый «буферный интерфейс», который напрямую дает доступ к собственному буферу памяти операциям, которые могут с ним работать. Запись бинарных данных — одна из таких операций.

Многие объекты также позволяют бинарным данным напрямую быть прочитанными в их память с помощью файлового метода `readinto()`. Например:

```
>>> import array
>>> a = array.array('i', [0, 0, 0, 0, 0, 0, 0, 0])
>>> with open('data.bin', 'rb') as f:
...     f.readinto(a)
...
16
>>> a
array('i', [1, 2, 3, 4, 0, 0, 0, 0])
>>>
```

Однако нужно принять все меры предосторожности при использовании этого приёма, поскольку он часто является платформозависимым и зависит от таких вещей, как размер слова, порядок следования байтов (big-endian или little-endian). См. рецепт 5.9., где приведён другой пример чтения бинарных данных в изменяемый (mutable) буфер.

5.5. Запись в файл, которого ещё нет

Задача

Вы хотите записать данные в файл, но только в том случае, если его ещё нет в файловой системе.

Решение

Эта задача легко решается с помощью использования малоизвестного режима `x` работы `open()` (вместо обычного режима `w`):

```
>>> with open('somefile', 'wt') as f:
...     f.write('Hello\n')
...
>>> with open('somefile', 'xt') as f:
...     f.write('Hello\n')
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'somefile'
>>>
```

Если файл в бинарном режиме, используйте режим *xb* вместо *xt*.

Обсуждение

Этот рецепт демонстрирует удивительно элегантное решение проблемы, иногда возникающей при записи в файлы (например, случайной перезаписи существующего файла). Альтернативное решение — предварительная проверка:

```
>>> import os
>>> if not os.path.exists('somefile'):
...     with open('somefile', 'wt') as f:
...         f.write('Hello\n')
... else:
...     print('File already exists!')
...
... File already exists!
>>>
```

Очевидно, что использование режима *x* намного проще. Важно отметить, что режим *x* доступен для функции *open()* только в Python 3. Этот режим не существовал в ранних версиях Python или низкоуровневых библиотеках на языке С, использованных в реализации Python.

5.6. Выполнение операций ввода-вывода над строками

Задача

Вы хотите скормить текст или бинарную строку программе, которая способна работать с файлоподобными объектами.

Решение

Используйте классы *io.StringIO()* и *io.BytesIO()* для создания файлоподобных объектов, которые могут работать со строковыми данными. Например:

```
>>> s = io.StringIO()
>>> s.write('Hello World\n')
12
>>> print('This is a test', file=s)
```

```
15
>>> # Получить все уже записанные данные
>>> s.getvalue()
'Hello World\nThis is a test\n'
>>>
```

```
>>> # Обернуть существующую строку файловым интерфейсом
>>> s = io.StringIO('Hello\nWorld\n')
>>> s.read(4)
'Hello'
>>> s.read()
'World\n'
```

Класс *io.StringIO* должен быть использован только для работы с текстом. Если вы работаете с бинарными данными, используйте *io.BytesIO*. Например:

```
>>> s = io.BytesIO()
>>> s.write(b'binary data')
>>> s.getvalue()
b'binary data'
>>>
```

Обсуждение

Классы *StringIO* и *BytesIO* наиболее полезны в случаях, когда вам нужно подменить обычный файл. Например, в юнит-тестах вы могли бы использовать *StringIO* для создания файлоподобного объекта, содержащего тестовые данные, которые скармливаются функции, предназначеннной для работы с файлами.

Обратите внимание, что экземпляры *StringIO* и *BytesIO* не имеют настоящего целочисленного файлового дескриптора. Поэтому они не будут работать с программами, которые требуют использования настоящих системных файлов (файлов, каналов, сокетов).

5.7. Чтение и запись сжатых файлов с данными

Задача

Вам нужно прочесть или записать данные в файл, сжатый *gzip* или *bz2*.

Решение

Модули *gzip* и *bz2* делают работу с такими файлами очень лёгкой. Оба модуля предоставляют альтернативную реализацию функции *open()*, которые могут быть использованы для этой цели. Например, чтобы прочесть сжатые файлы как текст, сделайте так:

```
# Сжатие с помощью gzip
import gzip
with gzip.open('somefile.gz', 'rt') as f:
    text = f.read()

# Сжатие с помощью bz2
import bz2
with bz2.open('somefile.bz2', 'rt') as f:
    text = f.read()
```

Как показано выше, весь ввод и вывод будет использовать текст и проводить кодирование/декодирование в Unicode. Если же вы хотите работать с бинарными данными, используйте файловые режимы *rb* или *wb*.

Обсуждение

Чтение и запись сжатых данных по большей части просты. Однако стоит знать, что выбор правильного файлового режима критически важен. Если вы не обозначите режим явно, то будет выбран режим по умолчанию, то есть бинарный, а это сломает программы, которые ожидают получить текст. *gzip.open()* и *bz2.open()* принимают те же параметры, что и встроенная функция *open()*, включая *encoding*, *errors*, *newline* и т.д.

При записи сжатых данных с помощью необязательного именованного аргумента *compresslevel* может быть установлен уровень компрессии. Например:

```
with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:
    f.write(text)
```

Уровень по умолчанию — это 9, то есть наивысший. Более низкие уровни увеличивают скорость, но снижают степень сжатия данных.

И последнее: малоизвестная особенность *gzip.open()* и *bz2.open()* заключается в том, что они могут работать уровнем выше существующего файла,

открытого в бинарном режиме. Например, такой код работает:

```
import

f = open('somefile.gz', 'rb')
with gzip.open(f, 'rt') as g:
    text = g.read()
```

Это позволяет модулям *gzip* и *bz2* работать с различными файлоподобными объектами, такими как сокеты, каналы и файлы в оперативной памяти.

5.8. Итерирование по записям фиксированного размера

Задача

Вместо того, чтобы итерировать по файлу построчно, вы хотите итерировать по коллекции записей фиксированного размера или кусочкам (чанкам).

Решение

Используйте функции *iter()* и *functools.partial()*, чтобы выполнить этот клёвый фокус:

```
from functools import partial

RECORD_SIZE = 32

with open('somefile.data', 'rb') as f:
    records = iter(partial(f.read, RECORD_SIZE), b'')
    for r in records:
        ...
```

Объект *records* в этом примере является итерируемым; он будет производить кусочки (чанки) фиксированного размера, пока не будет достигнут конец файла. Однако стоит отметить, что в последнем элементе может быть на несколько байтов меньше, чем ожидается, если размер файла не делится на точную длину размера записи.

Обсуждение

Малоизвестная возможность функции `iter()` заключается в том, что она может создать итератор, если вы передадите ей вызываемый объект и «значение-страж» (пороговое). Получившийся итератор просто снова и снова вызывает предоставленный вызываемый объект, пока он не вернет значение-страж, что приведёт к завершению итерирования.

В вышеприведённом решении `functools.partial` используется для создания вызываемого объекта, который читает фиксированное количество байтов из файла каждый раз, когда вызывается. Страж `b''` — то, что будет возвращено при попытке чтения файла, когда будет достигнут его конец.

И последнее: в показанном выше решении файл был открыт в бинарном режиме. Для чтений записей фиксированного размера это является наиболее распространенным случаем. В случае же текстовых файлов более распространенным будет построчное чтение (итератор выполняет его по умолчанию).

5.9. Чтение бинарных данных в изменяемый (мутабельный) буфер

Задача

Вы хотите читать бинарные данные непосредственно в изменяемый буфер без какого-либо промежуточного копирования. Возможно, вы хотите изменить данные на месте и записать их обратно в файл.

Решение

Чтобы прочесть данные в изменяемый массив, используйте файловый метод `readinto()`. Например:

```
import os.path

def read_into_buffer(filename):
    buf = bytearray(os.path.getsize(filename))
    with open(filename, 'rb') as f:
        f.readinto(buf)
    return buf
```

Вот пример использования:

```
>>> # Записываем файл примера
>>> with open('sample.bin', 'wb') as f:
...     f.write(b'Hello World')
...
>>> buf = read_into_buffer('sample.bin')
>>> buf
bytearray(b'Hello World')
>>> buf[0:5] = b'Hello'
>>> buf
bytearray(b'Hello World')
>>> with open('newsample.bin', 'wb') as f:
...     f.write(buf)
...
11
>>>
```

Обсуждение

Метод `readinto()` может быть использован для заполнения данными любого предварительно выделенного (preallocated) массива. Это даже включает массивы, созданные с помощью модуля `array` или библиотек типа `numpy`. В отличие от обычного метода `read()`, метод `readinto()` заполняет содержание текущего буфера вместо выделения и возвращения новых объектов. Так что вы можете использовать его, чтобы избежать излишних выделений памяти. Например, если вы читаете бинарный файл, состоящий из записей одинакового размера, вы можете написать такую программу:

```
record_size = 32      # Размер каждой записи (значение для подгонки)

buf = bytearray(record_size)
with open('somefile', 'rb') as f:
    while True:
        n = f.readinto(buf)
        if n < record_size:
            break
    # Используем содержимое буфера
    ...
```

Ещё одна интересная возможность — функция `memoryview()`, которая позволяет делать срезы [zero-copy](#) существующего буфера, и даже менять его содержимое. Например:

```
>>> buf
bytearray(b'Hello World')
```

```
>>> m1 = memoryview(buf)
>>> m2 = m1[-5:]
>>> m2
<memory at 0x100681390>
>>> m2[:] = b'WORLD'
>>> buf
bytearray(b'Hello WORLD')
>>>
```

При использовании метода `f.readinto()` нужно соблюдать осторожность: вы должны всегда проверять его код возврата, который является количеством фактически прочтённых байтов.

Если число байтов меньше размера предоставленного буфера, это может указывать на повреждение данных (например, если вы ожидали, что будет прочитано точное количество байтов).

И последнее: посмотрите на другие функции типа “`into`” в различных библиотечных модулях (например, `recv_into()`, `pack_into()` и т.д.) Многие другие компоненты Python имеют поддержку прямого ввода-вывода и доступа к данным, которая может быть использована для заполнения или изменения содержания массивов и буферов.

В [рецепте 6.12](#). приведён значительно более продвинутый пример интерпретации бинарных структур и использования представлений памяти (`memoryviews`).

5.10. Отображаемые в память бинарные файлы

Задача

Вы хотите отобразить в память бинарный файл в форме изменяемого массива байтов — вероятно, для произвольного доступа к его содержимому или изменений прямо на месте.

Решение

Используйте модуль `mmap` для отображения файлов в память. Вот полезная функция, с помощью которой можно открыть файл и отобразить его в память переносимым способом:

```
import os
import mmap

def memory_map(filename, access=mmap.ACCESS_WRITE):
    size = os.path.getsize(filename)
    fd = os.open(filename, os.O_RDWR)
    return mmap.mmap(fd, size, access=access)
```

Чтобы использовать эту функцию, вам нужен уже созданный и наполненный данными файл. Вот пример того, как вы можете сначала создать файл и увеличить его до нужного размера:

```
>>> size = 1000000
>>> with open('data', 'wb') as f:
...     f.seek(size-1)
...     f.write(b'\x00')
...
>>>
```

А вот пример отображения содержимого в память с помощью функции *memory_map()*:

```
>>> m = memory_map('data')
>>> len(m)
1000000
>>> m[0:10]
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> m[0]
0
>>> # Переписывание среза
>>> m[0:11] = b'Hello World'
>>> m.close()

>>> # Проверка того, что изменения были сделаны
>>> with open('data', 'rb') as f:
...     print(f.read(11))
...
b'Hello World'
>>>
```

Объект *mmap*, возвращаемый функцией *map()*, может быть также использован в качестве менеджера контекста. В этом случае отображённый файл закрывается автоматически. Например:

```
>>> with memory_map('data') as m:
...     print(len(m))
```

```
...     print(m[0:10])
...
1000000
b'Hello World'
>>> m.closed
True
>>>
```

По умолчанию показанная функция *memory_map()* открывает файл и на чтение, и на запись. Любые изменения данных копируются в исходный файл. Если требуется организовать доступ только для чтения, предоставьте *mmap.ACCESS_READ* в качестве аргумента *access*. Например:

```
m = memory_map(filename, mmap.ACCESS_READ)
```

Если вы намерены локально изменять данные, но не хотите, чтобы изменения записывались в исходный файл, используйте *mmap.ACCESS_COPY*:

```
m = memory_map(filename, mmap.ACCESS_COPY)
```

Обсуждение

Использование *mmap* для отображения файлов в память может стать элегантным и эффективным решением для произвольного доступа к содержимому файла. Например, вместо открытия файла и выполнения различных комбинаций вызовов *seek()*, *read()* и *write()*, вы просто отображаете файл и получаете доступ к любым данным через операции извлечения срезов.

Обычно память, выделяемая *mmap()*, выглядит как объект *bytearray*. Однако вы можете интерпретировать данные по-разному, используя функцию *memoryview*. Например:

```
>>> m = memory_map('data')
>>> # Представление памяти беззнаковых целых чисел
>>> v = memoryview(m).cast('I')
>>> v[0] = 7
>>> m[0:4]
b'\x07\x00\x00\x00'
>>> m[0:4] = b'\x07\x01\x00\x00'
>>> v[0]
263
>>>
```

Стоит отметить, что отображение файла в память не вызывает чтения файла в память целиком. Он не копируется в некий буфер памяти или массив. Вместо этого операционная система выделяет участок виртуальной памяти под содержимое файла. По мере того, как вы обращаетесь к различным участкам, эти куски файла будут читаться и отображаться в участок памяти по мере необходимости. Однако части файла, к которым никогда не производился доступ, останутся на диске.

Если не единственный интерпретатор Python отображает в память один и тот же файл, получившийся объект *ttbar* может быть использован для обмена данными между интерпретаторами. Интерпретаторы могут читать и записывать данные одновременно, и изменения, которые были сделаны в одном интерпретаторе, автоматически будут доступны в других. Очевидно, что синхронизация требует дополнительного внимания, но этот подход иногда используется в качестве альтернативы передаче данных через каналы или сокеты.

Показанный выше рецепт написан максимально обобщённо, он работает и в Windows, и в Unix. Однако стоит отметить, что есть специфические для каждой платформы отличия в том, как «под капотом» работает *ttbar()*. Также есть возможности по созданию анонимно отображенных участков памяти. Если вас это интересует, прочтите [соответствующий раздел документации Python](#).

5.11. Манипулирование путями к файлам

Задача

Вам нужно манипулировать путями к файлам, чтобы найти имя файла, название каталога, абсолютный путь и т.д.

Решение

Для работы с файловыми путями используйте функции из модуля *os.path*. Вот пример, который иллюстрирует несколько ключевых возможностей:

```
>>> import os  
>>> path = '/Users/beazley/Data/data.csv'  
  
>>> # Получение последнего компонента пути
```

```
>>> os.path.basename(path)
'data.csv'

>>> # Получение имени каталога
>>> os.path.dirname(path)
'/Users/beazley/Data'

>>> # Соединение компонентов пути
>>> os.path.join('tmp', 'data', os.path.basename(path))
'tmp/data/data.csv'

>>> # Раскрытие домашнего каталога пользователя
>>> path = '~/Data/data.csv'
>>> os.path.expanduser(path)
'/Users/beazley/Data/data.csv'

>>> # Отделение расширения файла
>>> os.path.splitext(path)
('~/Data/data', '.csv')
>>>
```

Обсуждение

Для любых манипуляций с именами файлов вы должны использовать модуль `os.path`, а не изобретать собственный велосипед из стандартных строковых операций. Во-первых, это важно для переносимости. Модуль `os.path` понимает различия между *Unix* и *Windows* и может надёжно работать с именами типа `Data/data.csv` и `Data\data.csv`. Во-вторых, вы не должны тратить время на велосипедостроение. Обычно лучше использовать готовые решения.

Стоит отметить, что в модуле `os.path` намного больше возможностей, чем показано в этом рецепте. Обратитесь к документации, чтобы узнать о функциях для тестирования файлов, работы с символическими ссылками и т.д.

5.12. Проверка существования файла

Задача

Вам нужно выяснить, существует ли файл или каталог.

Решение

Используйте `os.path`, чтобы проверить, существует ли файл или каталог.

Например:

```
>>> import os  
>>> os.path.exists('/etc/passwd')  
True  
>>> os.path.exists('/tmp/spam')  
False  
>>>
```

Вы можете выполнить дополнительные тесты, чтобы проверить тип файла.

Эти проверки возвращают `False`, если файл не существует:

```
>>> # Это обычный файл?  
>>> os.path.isfile('/etc/passwd')  
True  
  
>>> # Это каталог?  
>>> os.path.isdir('/etc/passwd')  
False  
  
>>> # Это символьская ссылка?  
>>> os.path.islink('/usr/local/bin/python3')  
True  
  
>>> # Получить прилинкованный файл  
>>> os.path.realpath('/usr/local/bin/python3')  
'/usr/local/bin/python3.3'  
>>>
```

Если вам нужно получить метаданные (например, размер или дату изменения файла), это тоже можно сделать с помощью модуля `os.path`:

```
>>> os.path.getsize('/etc/passwd')  
3669  
>>> os.path.getmtime('/etc/passwd')  
1272478234.0  
>>> import time  
>>> time.ctime(os.path.getmtime('/etc/passwd'))  
'Wed Apr 28 13:10:34 2010'  
>>>
```

Обсуждение

Проверка файлов с помощью `os.path` становится очень простой операцией. Единственное, о чем стоит помнить, так это о разрешениях — особенно при операциях получения метаданных. Например:

```
>>> os.path.getsize('/Users/guido/Desktop/foo.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/genericpath.py", line 49, in getsize
    return os.stat(filename).st_size
PermissionError: [Errno 13] Permission denied: '/Users/guido/Desktop/foc
>>>
```

5.13. Получение содержимого каталога

Задача

Вы хотите получить список файлов, содержащихся в каталоге файловой системы.

Решение

Используйте функцию `os.listdir()` для получения списка файлов в каталоге:

```
import os
names = os.listdir('somedir')
```

Вы получите «сырой» список содержимого каталога, включающий все файлы, подкаталоги, символические ссылки и т.п. Если вам нужно как-то отфильтровать эти данные, используйте генератор списков вместе с различными функциями библиотеки `os.path()`. Например:

```
import os.path
# Получить все обычные файлы
names = [name for name in os.listdir('somedir')
         if os.path.isfile(os.path.join('somedir', name))]

# Получить все каталоги
dirnames = [name for name in os.listdir('somedir')
            if os.path.isdir(os.path.join('somedir', name))]
```

Строковые методы `startswith()` и `endswith()` также могут быть полезны для

фильтрации содержимого каталога. Например:

```
pyfiles = [name for name in os.listdir('somedir')
           if name.endswith('.py')]
```

Для поиска совпадений по имени файла вы можете использовать модули *glob* или *fnmatch*. Например:

```
import glob
pyfiles = glob.glob('somedir/*.py')

from fnmatch import fnmatch
pyfiles = [name for name in os.listdir('somedir')
           if fnmatch(name, '*.py')]
```

Обсуждение

Получить содержимое каталога просто, но эта операция даёт вам просто имена элементов в каталоге. Если вы хотите получить дополнительные метаданные, такие как размеры файлов, даты изменений и т.д., вам нужны либо дополнительные функции модуля *os.path*, либо функция *os.stat()*.

Например:

```
# Пример получения содержимого каталога

import os
import os.path
import glob

pyfiles = glob.glob('*.*py')

# Получение размеров файлов и дат модификации
name_sz_date = [(name, os.path.getsize(name), os.path.getmtime(name))
                 for name in pyfiles]

for name, size, mtime in name_sz_date:
    print(name, size, mtime)

# Альтернатива: получение метаданных
file_metadata = [(name, os.stat(name)) for name in pyfiles]
for name, meta in file_metadata:
    print(name, meta.st_size, meta.st_mtime)
```

И последнее: в работе с именами файлов есть тонкие моменты, связанные с кодировками. Обычно записи, возвращаемые функциями типа *os.listdir()*,

декодируются согласно установленной по умолчанию в системе кодировке имен файлов. Однако возможно, что при некоторых обстоятельствах вам придётся столкнуться с недекодируемыми именами файлов. Рецепты 5.14. и 5.15. содержат дополнительные сведения о работе с такими именами.

5.14. Обход кодировки имен файлов

Задача

Вы хотите выполнить операции ввода-вывода, используя «сырые» имена файлов, которые не декодируются и не кодируются с помощью системной кодировки имён файлов по умолчанию.

Решение

По умолчанию все имена файлов кодируются и декодируются согласно кодировке, возвращаемой `sys.getfilesystemencoding()`. Например:

```
>>> sys.getfilesystemencoding()
'utf-8'
>>>
```

Если вы по какой-то причине хотите обойти эту кодировку, определите имя файла, используя «сырую» строку байтов. Например:

```
>>> # Запись файла с использованием имени в Unicode
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('Spicy!')
...
6
>>> # Содержимое каталога (декодировано)
>>> import os
>>> os.listdir('.')
['jalapeño.txt']

>>> # Содержимое каталога (сырое)
>>> os.listdir(b'.') # Заметьте: байтовая строка
[b'jalapen\xcc\x83o.txt']

>>> # Открыть файл с сырым именем
>>> with open(b'jalapen\xcc\x83o.txt') as f:
...     print(f.read())
...
```

Spicy!

>>>

Как вы можете видеть в двух последних операциях, обращение с именами файлов немного меняется, когда байтовые строки передаются связанным с файлами функциям, таким как `open()` и `os.listdir()`.

Обсуждение

В обычных обстоятельствах вам не нужно волноваться о кодировании и декодировании имён файлов — обычные операции с именами файлов «просто работают». Однако многие операционные системы могут позволить пользователю случайно или по злому умыслу создать файлы, которые не соответствуют ожидаемым правилам кодировки. Такие имена файлов могут загадочным образом сломать программы на Python, которые работают с большим количеством файлов.

Чтение каталогов и работа с именами файлов как сырьими недекодированными байтами может решить эту проблему, хотя и за счёт некоторых неудобств при программировании.

См. рецепт 5.15. о выводе недекодируемых имён файлов.

5.15. Вывод «плохих» имён файлов

Задача

Ваша программа получила список содержимого каталога, но когда она попыталась вывести эти имена файлов, то упала с исключением `UnicodeEncodeError` и загадочным сообщением “surrogates not allowed”.

Решение

При выводе имен файлов неизвестного происхождения, используйте преобразование для избежания ошибок:

```
def bad_filename(filename):
    return repr(filename)[1:-1]

try:
```

```
    print(filename)
except UnicodeEncodeError:
    print(bad_filename(filename))
```

Обсуждение

Этот рецепт решает редкую, но очень раздражающую проблему, которая касается программ, работающих с файловой системой. По умолчанию Python предполагает, что все имена файлов закодированы согласно установке, которая возвращается функцией `sys.getfilesystemencoding()`. Однако некоторые файловые системы не заставляют соблюдать это ограничение, позволяя создавать файлы с неправильной кодировкой. Это не частый случай, но всё же есть опасность, что некий пользователь сделает что-то глупое и случайно создаст такой файл (например, передаст неправильное имя файла функции `open()` в какой-то забагованной программе).

При выполнении команд типа `os.listdir()` неправильные имена файлов загоняют Python в безвыходную ситуацию. С одной стороны, он не может просто отбросить неправильное имя. С другой стороны, он не может превратить имя файла в правильную текстовую строку. Python действует так: берет недекодируемое байтовое значение `\xhh` в имени файла и отображает его в так называемую «суррогатную кодировку», представленную символом Unicode `\udchh`. Вот пример того, как неправильный список содержимого каталога может выглядеть, если он содержит имя файла `bäd.txt`, закодированное в Latin-1 вместо UTF-8:

```
>>> import os
>>> files = os.listdir('.')
>>> files
['spam.py', 'b\udce4d.txt', 'foo.txt']
>>>
```

Если у вас есть код, который манипулирует именами файлов или даже передает их функциям (таким как `open()`), всё работает нормально. Вы попадете в неприятности только в ситуациях, где вы хотите вывести имя файла (вывод, логирование и т.п.) Ваша программа упадет, если вы захотите вывести показанный выше список:

```
>>> for name in files:
...     print(name)
... 
```

```
spam.py
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udce4' in
position 1: surrogates not allowed
>>>
```

Причина падения в том, что символ \udce4 не является валидным в Unicode. Это вторая половина двухсимвольной комбинации, известной как «суррогатная пара». Поскольку первая часть отсутствует, это не валидный Unicode. Поэтому единственный способ успешно произвести вывод — предпринять корректирующее действие, если встретится неправильное имя файла. Например:

```
>> for name in files:
...     try:
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
b\udce4d.txt
foo.txt
>>>
```

Выбор того, что будет делать функция *bad_filename()*, во многом зависит от вас. Другая возможность — как-то перекодировать значение:

```
def bad_filename(filename):
    temp = filename.encode(sys.getfilesystemencoding(), \
                          errors='surrogateescape')
    return temp.decode('latin-1')
```

При использовании этой версии вы получите следующее:

```
>>> for name in files:
...     try:
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
b d.txt
foo.txt
>>>
```

Этот рецепт наверняка будет проигнорирован большинством читателей. Однако если вы пишете критически важные скрипты, которым нужно надёжно работать с именами файлов и файловой системой, об этом стоит подумать. В противном случае вы можете столкнуться с ситуацией, когда вам придется тащиться в офис на выходных и сражаться с какой-то непонятной ошибкой.

5.16. Добавление или изменение кодировки уже открытого файла

Задача

Вы хотите добавить или изменить кодировку Unicode уже открытого файла, не закрывая его.

Решение

Если вы хотите добавить кодирование/декодирование в Unicode уже существующему файловому объекту, открытому в бинарном режиме, оберните его объектом *io.TextIOWrapper()*. Например:

```
import urllib.request
import io

u = urllib.request.urlopen('http://www.python.org')
f = io.TextIOWrapper(u,encoding='utf-8')
text = f.read()
```

Если вы хотите изменить кодировку файла, открытого в текстовом режиме, используйте метод *detach()* для удаления существующего слоя текстовой кодировки перед заменой его новым. Вот пример изменения кодировки в *sys.stdout*:

```
>>> import sys
>>> sys.stdout.encoding
'UTF-8'
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='latin-1')
>>> sys.stdout.encoding
'latin-1'
>>>
```

Если вы это сделаете, то можете «поломать» вывод вашего терминала.

Приведенный выше пример — это просто иллюстрация подхода.

Обсуждение

Система ввода-вывода построена на последовательности слоёв. Вы можете увидеть эти слои, если попробуете сделать следующее:

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f.buffer
<_io.BufferedWriter name='sample.txt'>
>>> f.buffer.raw
<_io.FileIO name='sample.txt' mode='wb'>
>>>
```

В этом примере *io.TextIOWrapper* — это слой для обработки текста, который кодирует и декодирует в Unicode, *ioBufferedWriter* — буферизированный слой ввода-вывода, который работает с бинарными данными, а *ioFileIO* — «сырой файл», представляющий низкоуровневый файловый дескриптор в операционной системе. Добавление или изменение текстовой кодировки вовлекает добавление и изменение только верхнего слоя — *io.TextIOWrapper*.

Общее правило: небезопасно напрямую манипулировать слоями, используя показанные выше атрибуты. Например, вот что произойдет, если вы попытаетесь изменить кодировку с помощью этого приёма:

```
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>> f.write('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

Это не работает, поскольку изначальное значение *f* было уничтожено, а лежащий в основе файл закрыт.

Метод *detach()* отделяет верхний слой файла и возвращает следующий, более низкоуровневый слой. Далее высший слой уже нельзя использовать.

Например:

```
>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> b = f.detach()
>>> b
<_io.BufferedWriter name='sample.txt'>
>>> f.write('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: underlying buffer has been detached
>>>
```

После отделения, однако, вы можете добавить новый верхний слой возвращаемому результату. Например:

```
>>> f = io.TextIOWrapper(b, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>>
```

Хотя здесь мы показали изменение кодировки, этот приём может быть использован для изменения обработки строк, политики обработки ошибок и других аспектов работы с файлами. Например:

```
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='ascii',
...     errors='xmlcharrefreplace')
>>> print('Jalape\u00f1o')
Jalape&#241;o
>>>
```

Отметьте, как не входящий в ASCII символ ю был заменён на \њ в выводе.

5.17. Запись байтов в текстовый файл

Задача

Вы хотите записать сырье байты в файл, открытый в текстовом режиме.

Решение

Просто запишите байтовые данные в *buffer*. Например:

```
>>> import sys
>>> sys.stdout.write(b'Hello\n')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>> sys.stdout.buffer.write(b'Hello\n')
Hello
5
>>>
```

Похожим образом бинарные данные могут быть прочитаны из текстового файла путём чтения из атрибута *buffer*.

Обсуждение

Система ввода-вывода построена на слоях. Текстовые файлы конструируются путём добавления слоя кодирования/декодирования в Unicode поверх буферизованного файла, открытого в бинарном режиме. Атрибут *buffer* просто ссылается на этот файл. Если вы обращаетесь к нему, то обходите слой текстового кодирования/декодирования.

Пример с *sys.stdout* может быть рассмотрен как особый случай. По умолчанию *sys.stdout* всегда открывается в текстовом режиме. Однако если вы пишете скрипт, которому на самом деле нужно сбрасывать бинарные данные в стандартный вывод, вы можете использовать показанный приём для обхода текстовой кодировки.

5.18. Оборачивание существующего дескриптора файла для использования в качестве объекта файла

Задача

У вас есть целочисленный файловый дескриптор, соответствующий уже открытому каналу ввода-вывода операционной системы (например, файлу, каналу, сокету и т.п.), и вы хотите обернуть его высокоуровневым объектом файла Python.

Решение

Файловый дескриптор отличается от обычного открытого файла тем, что это просто целочисленный идентификатор, назначенный операционной системой, чтобы ссылаться на какой-то системный канал ввода-вывода. Если у вас есть дескриптор, вы можете обернуть его файловым объектом Python с помощью функции `open()`. Вы просто предоставляете целочисленный файловый дескриптор в качестве первого аргумента (вместо имени файла). Например:

```
# Открыть низкоуровневый файловый дескриптор
import os
fd = os.open('somefile.txt', os.O_WRONLY | os.O_CREAT)

# Превратить в настоящий файл
f = open(fd, 'wt')
f.write('hello world\n')
f.close()
```

Когда высокоуровневый файловый объект закрывается или разрушается, его файловый дескриптор тоже будет закрыт. Если это нежелательное поведение, передайте необязательный аргумент `closefd=False` функции `open()`. Например:

```
# Создать файловый объект, но не закрывать дескриптор по завершению
f = open(fd, 'wt', closefd=False)
...
```

Обсуждение

В Unix этот приём обравчивания файлового дескриптора может быть удобным способом для подключения файлоподобного интерфейса на существующий канал ввода-вывода, открытый другим способом (например, каналы, сокеты и т.д.) Вот пример с использованием сокетов:

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(client_sock, addr):
    print('Got connection from', addr)

    # Создать файловые обёртки текстового режима для чтения/записи в сокет
    client_in = open(client_sock.fileno(), 'rt', encoding='latin-1',
                     closefd=False)
    client_out = open(client_sock.fileno(), 'wt', encoding='latin-1',
```

```

        closefd=False)

# Отправить клиенту эхом строки, используя файловый ввод-вывод
for line in client_in:
    client_out.write(line)
    client_out.flush()
client_sock.close()

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_client(client, addr)

```

Важно подчеркнуть, что вышеприведённый пример нужен только для иллюстрации возможностей встроенной функции `open()`, и он работает только в Unix. Если вы пытаетесь накрутить файлоподобный интерфейс на сокет, и хотите, чтобы ваш код был кроссплатформенным, используйте метод сокетов `makefile()`. Однако если переносимость вас не беспокоит, вы можете обнаружить, что представленное выше решение обладает намного большей производительностью, нежели `makefile()`.

Вы также можете использовать этот приём для создания псевдонима, который позволит уже открытому файлу использоваться немного другим способом, нежели тот, каким он был изначально открыт. Например, вот так вы можете создать файловый объект, который позволит выводить бинарные данные в `stdout` (который по умолчанию открыт в текстовом режиме):

```

import sys
# Создать файл в бинарном режиме для stdout
bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
bstdout.write(b'Hello World\n')
bstdout.flush()

```

Хотя можно обернуть существующий файловый дескриптор и использовать его в качестве настоящего файла, обратите внимание, что не все файловые режимы могут поддерживаться, и что некоторые типы файловых дескрипторов могут обладать забавными побочными эффектами (особенно по отношению к обработке ошибок, условиям достижения конца файла и т.д.). Это поведение также может изменяться в зависимости от операционной системы. В частности, ни один из приведённых примеров не будет работать за пределами Unix. Суть в том, что вам нужно тщательно

тестировать свою реализацию, чтобы убедиться в том, что она работает так, как вы ожидаете.

5.19. Создание временных файлов и каталогов

Задача

Вам нужно создать временный файл или каталог, которые будут использоваться во время выполнения вашей программы. После, возможно, вы захотите, чтобы они были удалены.

Решение

В модуле *tempfile* есть различные функции, которые помогут решить эту задачу. Чтобы создать безымянный временный файл, используйте *tempfile.TemporaryFile*:

```
from tempfile import TemporaryFile

with TemporaryFile('w+t') as f:
    # Чтение/запись в файл
    f.write('Hello World\n')
    f.write('Testing\n')

    # Перейти в начало и прочесть данные
    f.seek(0)
    data = f.read()

# Временный файл уничтожен
```

Также вы можете использовать файл таким образом:

```
f = TemporaryFile('w+t')
# Использовать временный файл
...
f.close()
# Файл уничтожен
```

Первый аргумент, передаваемый в *TemporaryFile()*, это режим файла: обычно для текстовых файлов это *w+t*, а для бинарных — *w+b*. Этот режим одновременно поддерживает чтение и запись, что в данном случае полезно,

поскольку закрытие файла для смены режима разрушило бы его. *TemporaryFile()* дополнительно принимает те же аргументы, что и встроенная функция *open()*. Например:

```
with TemporaryFile('w+t', encoding='utf-8', errors='ignore') as f:  
    ...
```

На большинстве Unix-систем файл, созданный функцией *TemporaryFile()*, является безымянным и даже не имеет местоположения в каталоге. Если вы хотите преодолеть это ограничение, используйте *NamedTemporaryFile()*.

Например:

```
from tempfile import NamedTemporaryFile  
  
with NamedTemporaryFile('w+t') as f:  
    print('filename is:', f.name)  
    ...  
  
# Файл автоматически уничтожается
```

Здесь атрибут *f.name* открытого файла содержит имя временного файла. Это может быть полезно, если оно передается какой-то другой программе, которой нужно будет открыть этот файл. Как и в случае *TemporaryFile()*, получившийся файл будет автоматически уничтожен после закрытия. Если вы не хотите, чтобы это произошло, передайте в функцию именованный аргумент *delete=False*. Например:

```
with NamedTemporaryFile('w+t', delete=False) as f:  
    print('filename is:', f.name)  
    ...
```

Чтобы создать временный каталог, используйте *tempfile.TemporaryDirectory()*. Например:

```
from tempfile import TemporaryDirectory  
with TemporaryDirectory() as dirname:  
    print('dirname is:', dirname)  
    # Использовать каталог  
    ...  
# Каталог и всё его содержимое уничтожается
```

Обсуждение

Функции *TemporaryFile()*, *NamedTemporaryFile()* и *TemporaryDirectory()* – вероятно, самый удобный способ работы с временными файлами и каталогами, потому что они автоматически управляются со всеми шагами создания и последующей чистки. На низком уровне вы также можете использовать *mkstemp()* и *mkdtemp()* для создания временных файлов и каталогов:

```
>>> import tempfile  
>>> tempfile.mkstemp()  
(3, '/var/folders/7W/7Wz15sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp7fefhv')  
>>> tempfile.mkdtemp()  
'/var/folders/7W/7Wz15sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp5wvcv6'  
>>>
```

Однако эти функции не заботятся о последующем управлении. Например, функция *mkstemp()* просто возвращает сырой файловый дескриптор операционной системы и оставляет всю работу по превращению его в настоящий файл вам. Похожим образом вам нужно самостоятельно удалять файлы.

Обычно временные файлы создаются в определенном операционной системой месте сохранения по умолчанию, такому как */var/tmp* и т.п. Чтобы получить путь, используйте функцию *tempfile.gettempdir()*. Например:

```
>>> tempfile.gettempdir()  
'/var/folders/7W/7Wz15sfZEF0pljrEB1UMWE+++TI/-Tmp-'  
>>>
```

Все функции, связанные с временными файлами, позволяют вам менять этот каталог и принципы именования файлов с помощью именованных аргументов *prefix*, *suffix* и *dir*. Например:

```
>>> f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/tmp')  
>>> f.name  
'/tmp/mytemp8ee899.txt'  
>>>
```

И последнее: модуль *tempfile()* создает временные файлы наиболее безопасным способом из всех возможных. Это включает предоставление доступа только текущему пользователю и предпринятие шагов, предотвращающих состояние гонки (*race condition*) при создании файлов. Однако стоит знать, что на различных платформах этот модуль работает по-

разному. Чтобы уточнить своё понимание, обратитесь к официальной [документации](#).

5.20. Работа с последовательными портами

Задача

Вы хотите читать и записывать данные в последовательный порт. Обычно это нужно для взаимодействия с каким-то устройством (например, роботом или сенсором).

Решение

Хотя вы могли бы сделать это напрямую, используя примитивы ввода-вывода Python, лучшим выбором для последовательного взаимодействия является пакет [pySerial](#). Начать работать с пакетом очень легко. Вы просто открываете последовательный порт:

```
import serial
ser = serial.Serial('/dev/tty.usbmodem641',
                     baudrate=9600,
                     bytesize=8,
                     parity='N',
                     stopbits=1)
```

Имя устройства меняется в зависимости от его типа и операционной системы. Например, в Windows вы можете использовать устройство 0, 1 и так далее, чтобы открыть такие порты, как COM0 и COM1. Когда они открыты, вы можете читать и записывать данные, используя вызовы `read()`, `readline()` и `write()`. Например:

```
ser.write(b'G1 X50 Y50\r\n')
resp = ser.readline()
```

По большей части простой последовательный обмен данными весьма незамысловат.

Обсуждение

Простая на первый взгляд, последовательная коммуникация всё же может быть достаточно запутанной. Причина использовать пакеты типа pySerial в том, что они поддерживают продвинутые возможности (например, таймауты, контроль потока, сбросывание буфера, хендшейкинг и т.п.) Например, если вы хотите включить RTS-CTS-хендшейкинг, просто передайте `Serial()` аргумент `rtscts=True`. У пакета отличная документация, поэтому нет смысла её здесь пересказывать.

Помните, что весь ввод-вывод с использованием последовательных портов является бинарным. Поэтому убедитесь, что ваша программа использует байты, а не текст (или производит правильное кодирование/декодирование). Модуль `struct` может также оказаться полезным, если вам нужно будет создавать бинарные команды или пакеты.

5.21. Сериализация объектов Python

Задача

Вам нужно сериализовать объект Python в поток байтов, чтобы вы смогли сохранить их в файл или базу данных, или же передать их по сети.

Решение

Наиболее распространённый подход к сериализации данных — это использование модуля `pickle`. Чтобы сохранить объект в файл, сделайте так:

```
import pickle

data = ... # Какой-то объект Python
f = open('somefile', 'wb')
pickle.dump(data, f)
```

Чтобы сохранить объект в строку, используйте `pickle.dumps()`:

```
s = pickle.dumps(data)
```

Чтобы воссоздать объект из потока байтов (byte stream), используйте либо `pickle.load()`, либо `pickle.loads()`. Например:

```
# Восстановление из файла
```

```
f = open('somefile', 'rb')
data = pickle.load(f)

# Восстановление из строки
data = pickle.loads(s)
```

Обсуждение

Для большинства программ использование функций *dump()* и *load()* — всё, что требуется от модуля *pickle*. Он «просто работает» — с большинством типов данных Python и экземплярами ваших собственных классов. Если вы работаете с какой-либо библиотекой, которая позволяет вам делать такие вещи как сохранение и восстановление объектов Python в базах данных или передача объектов по сети, то очень велик шанс, что именно *pickle* используется для этого.

pickle — это самоописывающаяся кодировка данных, специфическая для Python. Под самоописыванием мы подразумеваем, что сериализованные данные содержат информацию о начале и конце каждого объекта, а также и информацию об их типе. Поэтому вам не нужно переживать об определении формата записей — всё работает «из коробки». Например, при работе с несколькими объектами вы можете сделать так:

```
>>> import pickle
>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>
```

Вы можете сериализовать функции, классы и экземпляры, однако получающиеся данные кодируют только имена ссылок на связанные объекты кода. Например:

```
>>> import math
>>> import pickle
```

```
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
>>>
```

Когда данные десериализуются, то предполагается, что требуемый источник доступен. Модули, классы и функции будут автоматически импортированы при необходимости. Для приложений, где данные Python разделяются между интерпретаторами или разными компьютерами, это потенциально может оказаться проблемой, поскольку все машины должны иметь доступ к одному и тому же исходному коду.

`pickle.load()` никогда нельзя использовать на данных из непроверенных источников. В качестве побочного эффекта загрузки, `pickle` автоматически загрузит модули и создаст экземпляры. Однако злоумышленник, который знает принцип работы `pickle`, может создать специальные данные, которые заставят Python выполнить произвольные системные команды. Так что `pickle` можно использовать только внутренними данными и интерпретаторами, которые могут каким-то образом проводить аутентификацию друг друга.

Некоторые типы объектов не могут быть сериализованы. Это обычно те объекты, которые используют некоторое внешнее системное состояние — такие как открытые файлы, открытые сетевые соединения, потоки, процессы, фреймы стека и т.д. Определенные пользователем классы могут иногда обойти эти ограничения, предоставляя методы `__getstate__()` и `__setstate__()`. Если они определены, `pickle.dump()` вызовет `__getstate__()`, чтобы получить объект, пригодный для сериализации. Похожим образом `__setstate__()` будет вызыван при десериализации. Чтобы проиллюстрировать возможности, ниже приведён класс, который внутри определяет поток, но при этом может быть сериализован и десериализован:

```
# countdown.py
import time
import threading

class Countdown:
    def __init__(self, n):
        self.n = n
        self.thr = threading.Thread(target=self.run)
        self.thr.daemon = True
        self.thr.start()

    def run(self):
        while self.n > 0:
```

```
    print('T-minus', self.n)
    self.n -= 1
    time.sleep(5)

def __getstate__(self):
    return self.n

def __setstate__(self, n):
    self.__init__(n)
```

Попробуйте применить *pickle*:

```
>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...
>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()
```

Теперь выйдите из Python и после перезапуска попробуйте вот это:

```
>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...
```

Вы должны увидеть, как поток волшебным образом возрождается к жизни, поднимаясь на том же месте, где он был, когда вы его сериализовали.

pickle не особенно эффективен для сериализации крупных структур данных, таких как бинарные массивы, созданные библиотеками типа *numpy* или модуля *array*. Если вы перемещаете большие объемы данных в массивах туда-сюда, вам лучше просто сохранять массивы в файлы или использовать более стандартизованную кодировку, такую как HDF5 (поддерживается не входящими в поставку Python библиотеками).

pickle по своей природе привязан к Python и исходному коду, поэтому вам не стоит использовать его для долговременного хранения данных. Например,

если исходный код изменится, все ваши сохранённые данные могут поломаться и стать нечитаемыми. Если честно, для хранения данных в базах данных и архивных хранилищах, вам лучше использовать более стандартные кодировки, такие как XML, CSV или JSON. Они поддерживаются большим количеством языков программирования, и более адаптируемы к изменениям в вашем исходном коде.

И последнее: стоит помнить, что у *pickle* огромное количество различных параметров и хитрых случаев применения. В большинстве обычных ситуаций вам не нужно о них волноваться, но если вы создаете серьёзное приложение, использующее *pickle* для сериализации, не забудьте прочитать [официальную документацию](#).

6. Кодирование и обработка данных

Основная тема этой главы — использование Python для обработки данных, представленных в различных типах распространенных форматов, таких как файлы CSV, JSON, XML и упакованные бинарные записи. В отличие от главы о структурах данных, здесь мы не будем фокусироваться на конкретных алгоритмах, а рассмотрим задачу получения данных из программы и передачи данных в программу.

6.1. Чтение и запись данных в формате CSV

Задача

Вы хотите прочесть или записать данные в CSV-файл.

###Решение Для большей части CSV-данных можно использовать библиотеку csv. Предположим, например, что у вас есть данные о рынке акций в файле stocks.csv:

```
Symbol,Price,Date,Time,Change,Volume  
"AA",39.48,"6/11/2007","9:36am",-0.18,181800  
"AIG",71.38,"6/11/2007","9:36am",-0.15,195500
```

```
"AXP", 62.58, "6/11/2007", "9:36am", -0.46, 935000  
"BA", 98.31, "6/11/2007", "9:36am", +0.12, 104800  
"C", 53.08, "6/11/2007", "9:36am", -0.25, 360900  
"CAT", 78.29, "6/11/2007", "9:36am", -0.23, 225400
```

Вот как вы могли бы прочитать данные в последовательность кортежей:

```
import csv  
with open('stocks.csv') as f:  
    f_csv = csv.reader(f)  
    headers = next(f_csv)  
    for row in f_csv:  
        # Обработка строки  
        ...
```

В приведённом выше коде строке соответствует кортеж. Поэтому для доступа к определенному полю вам нужно использовать индексирование: `row[0]` — Symbol, `row[4]` — Change.

Поскольку такое индексирование часто может быть запутанным, вы можете захотеть использовать именованные кортежи. Например:

```
from collections import namedtuple  
with open('stock.csv') as f:  
    f_csv = csv.reader(f)  
    headings = next(f_csv)  
    Row = namedtuple('Row', headings)  
    for r in f_csv:  
        row = Row(*r)  
        # Обработка строки  
        ...
```

Это позволит использовать вместо индексов заголовки колонок, такие как `row.Symbol` и `row.Change`. Стоит отметить, что это сработает только в том случае, если заголовки колонок являются валидными идентификаторами Python. Если это не так, вы должны будете обработать эти заголовки (например, заменить неподходящие символы подчёркиваниями и т.п.)

Еще одна альтернатива — прочесть данные в последовательность словарей. Чтобы это сделать, используйте такой код:

```
import csv  
with open('stocks.csv') as f:  
    f_csv = csv.DictReader(f)  
    for row in f_csv:
```

```
# Обработка строки
```

```
...
```

В этой версии вы можете обращаться к элементом каждой строки, используя заголовки строки. Например, `row['Symbol']` или `row['Change']`.

Чтобы записать данные в CSV, вы также можете использовать модуль csv, но создавая объект `writer`. Например:

```
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [ ('AA', 39.48, '6/11/2007', '9:36am', -0.18, 181800),
        ('AIG', 71.38, '6/11/2007', '9:36am', -0.15, 195500),
        ('AXP', 62.58, '6/11/2007', '9:36am', -0.46, 935000),
    ]

with open('stocks.csv', 'w') as f:
    f_csv = csv.writer(f)
    f_csv.writerow(headers)
    f_csv.writerows(rows)
```

Если у вас есть данные в форме последовательности словарей, сделайте так:

```
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [{ 'Symbol': 'AA', 'Price': 39.48, 'Date': '6/11/2007',
          'Time': '9:36am', 'Change': -0.18, 'Volume': 181800},
        { 'Symbol': 'AIG', 'Price': 71.38, 'Date': '6/11/2007',
          'Time': '9:36am', 'Change': -0.15, 'Volume': 195500},
        { 'Symbol': 'AXP', 'Price': 62.58, 'Date': '6/11/2007',
          'Time': '9:36am', 'Change': -0.46, 'Volume': 935000},
    ]

with open('stocks.csv', 'w') as f:
    f_csv = csv.DictWriter(f, headers)
    f_csv.writeheader()
    f_csv.writerows(rows)
```

Обсуждение

Вы должны практически всегда предпочитать модуль csv ручному разрезанию и парсингу CSV-данных. Например, вы можете подумывать написать такой код:

```
with open('stocks.csv') as f:
    for line in f:
        row = line.split(',')
```

```
# Обработка строки
```

```
...
```

Проблема такого подхода в том, что придётся разбираться с надоедливыми деталями. Например, если одно из полей окружено кавычками, вы должны будете их срезать. А если это закавыченное поле содержит запятую, код сломается, поскольку выдаст строку неверного размера.

По умолчанию библиотека csv запрограммирована понимать правила кодирования CSV, которые используются Microsoft Excel. Это, вероятно, наиболее распространенный вариант, и он с высокой вероятностью обеспечит вам наилучшую совместимость. Однако вы можете свериться с документацией модуля csv, и там вы найдете настройки, которые помогут работать с кодировками другого формата (например, заменить символ-разделитель и т.п.) Например, если вы хотите прочесть данные, разделенные символами табуляции, используйте вот такой код:

```
# Пример чтения разделенных символом табуляции значений
with open('stock.tsv') as f:
    f_tsv = csv.reader(f, delimiter='\t')
    for row in f_tsv:
        # Обработка строки
    ...

```

Если вы читаете данные в CSV и конвертируете их в именованные кортежи, вам нужно быть аккуратными с валидацией заголовков колонок. Например, CSV-файл может иметь строку заголовка, содержащую невалидный символ:

```
Street Address,Num-Premises,Latitude,Longitude
5412 N CLARK,10,41.980262,-87.668452
```

Это при создании экземпляра `namedtuple` возбудит исключение `ValueError`. Чтобы обойти проблему, вам может потребоваться почистить заголовки. Например, разобраться с невалидными символами с помощью регулярного выражения:

```
import re
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headers = [re.sub('[^a-zA-Z_]', '_', h) for h in next(f_csv)]
    Row = namedtuple('Row', headers)
    for r in f_csv:
        row = Row(*r)
```

```
# Обработка строки
```

```
...
```

Важно отметить, что модуль `csv` не пытается интерпретировать данные или конвертировать их в какой-то другой тип, нежели строку. Если такие преобразования нужны, их вам придется выполнить самостоятельно. Вот пример выполнения дополнительных преобразований типов CSV-данных:

```
col_types = [str, float, str, str, float, int]
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Применение преобразований к элементам строки
        row = tuple(convert(value) for convert, value in zip(col_types,
...
```

Альтернативный пример преобразования выбранных полей словарей:

```
print('Reading as dicts with type conversion')
field_types = [ ('Price', float),
                 ('Change', float),
                 ('Volume', int) ]

with open('stocks.csv') as f:
    for row in csv.DictReader(f):
        row.update((key, conversion(row[key])))
            for key, conversion in field_types)
    print(row)
```

В общем, вам стоит быть осторожными с такими преобразованиями. В реальном мире в CSV-файлах часто попадаются отсутствующие поля, повреждённые данные и прочие проблемы, которые могут поломать преобразования типов. Так что если ваши данные не являются гарантированно безошибочными, об этом стоит помнить (например, вы можете добавить подходящую обработку исключений).

Наконец, если ваша цель — чтение CSV-данных для выполнения анализа данных и статистических расчётов, вы можете взглянуть на пакет [Pandas](#). *Pandas* включает удобную функцию `pandas.read_csv()`, которая загружает CSV-данные в объект `DataFrame`. Далее вы можете провести различные статистические расчёты, отфильтровать данные и выполнить другие высокоуровневые операции. Пример вы можете найти в [рецепте 6.13](#).

6.2. Чтение и запись в формате JSON

Задача

Вы хотите прочитать или записать данные, закодированные в JSON (JavaScript Object Notation).

Решение

Модуль `json` предоставляет простой способ кодировать и декодировать данные в JSON. Две главные функции — `json.dumps()` и `json.loads()` — соответствуют интерфейсу других библиотек для сериализации, таких как `pickle`. Вот как вы можете превратить структуру данных Python в JSON:

```
import json

data = {
    'name' : 'ACME',
    'shares' : 100,
    'price' : 542.23
}

json_str = json.dumps(data)
```

А вот как можно превратить строку в JSON обратно в структуру данных Python:

```
# Запись JSON-данных
with open('data.json', 'w') as f:
    json.dump(data, f)

# Чтение данных
with open('data.json', 'r') as f:
    data = json.load(f)
```

Обсуждение

Кодирование в JSON поддерживает базовые типы: `None`, `bool`, `int`, `float`, `str`, а также списки, кортежи и словари, содержащие эти типы. В случае словарей ключами должны быть строки (все нестроковые ключи будут преобразованы в строки во время кодирования). Чтобы соответствовать спецификации

JSON, вы должны кодировать только списки и словари Python. Более того, для веб-приложений стандартной практикой является использование именно словарей в качестве объектов верхнего уровня.

Формат JSON практически идентичен синтаксису Python, за исключением нескольких небольших изменений. Например, *True* отображается на *true*, *False* — на *false*, а *None* — на *null*. Вот пример того, как выглядят закодированные данные:

```
>>> json.dumps(False)
'false'
>>> d = {'a': True,
...        'b': 'Hello',
...        'c': None}
>>> json.dumps(d)
'{"b": "Hello", "c": null, "a": true}'
>>>
```

Если вы хотите изучить данные, которые вы раскодировали из JSON, часто бывает трудно установить их структуру путём простого вывода — особенно если присутствует многоуровневая вложенность или большое количество различных полей. Чтобы справиться с этой задачей, попробуйте функцию *pprint()* из модуля *pprint*. Она расположит ключи в алфавитном порядке и выведет словарь в более понятном виде. Вот пример того, как вы можете симпатично вывести результаты поиска по Twitter:

```
>>> from urllib.request import urlopen
>>> import json
>>> u = urlopen('http://search.twitter.com/search.json?q=python&rpp=5')
>>> resp = json.loads(u.read().decode('utf-8'))
>>> from pprint import pprint
>>> pprint(resp)
{'completed_in': 0.074,
 'max_id': 264043230692245504,
 'max_id_str': '264043230692245504',
 'next_page': '?page=2&max_id=264043230692245504&q=python&rpp=5',
 'page': 1,
 'query': 'python',
 'refresh_url': '?since_id=264043230692245504&q=python',
 'results': [{['created_at': 'Thu, 01 Nov 2012 16:36:26 +0000',
            'from_user': ...},
             {['created_at': 'Thu, 01 Nov 2012 16:36:14 +0000',
            'from_user': ...},
             {['created_at': 'Thu, 01 Nov 2012 16:36:13 +0000',
            'from_user': ...}]}]
```

```
'from_user': ...  
},  
{'created_at': 'Thu, 01 Nov 2012 16:36:07 +0000',  
 'from_user': ...  
}  
{'created_at': 'Thu, 01 Nov 2012 16:36:04 +0000',  
 'from_user': ...  
}],  
'results_per_page': 5,  
'since_id': 0,  
'since_id_str': '0'}  
>>>
```

В обычном случае декодирование JSON создаст из предоставленных данных словари или списки. Если вы хотите создать другие объекты, передайте *objects_pair_hook* или *object_hook* функции *json.loads()*. Например, вы можете декодировать JSON-данные, сохраняя их порядок в *OrderedDict*:

```
>>> s = '{"name": "ACME", "shares": 50, "price": 490.1}'  
>>> from collections import OrderedDict  
>>> data = json.loads(s, object_pairs_hook=OrderedDict)  
>>> data  
OrderedDict([('name', 'ACME'), ('shares', 50), ('price', 490.1)])  
>>>
```

Вот как вы можете превратить словарь JSON в объект Python:

```
>>> class JSONObject:  
...     def __init__(self, d):  
...         self.__dict__ = d  
...  
>>>  
>>> data = json.loads(s, object_hook=JSONObject)  
>>> data.name  
'ACME'  
>>> data.shares  
50  
>>> data.price  
490.1  
>>>
```

В последнем примере созданный при декодировании JSON-данных словарь передается как единственный аргумент в *__init__()*. Далее вы можете использовать его, как хотите, в том числе и напрямую в качестве экземпляра словаря объекта.

Есть несколько параметров, которые могут быть полезны при кодировании в JSON. Если вы хотите, чтобы вывод был симпатично отформатирован, вы можете использовать аргумент *indent* функции *json.dumps()*. В этом случае вывод будет красиво выводиться — в формате, похожем на вывод функции *pprint()*. Например:

```
>>> print(json.dumps(data))
{"price": 542.23, "name": "ACME", "shares": 100}
>>> print(json.dumps(data, indent=4))
{
    "price": 542.23,
    "name": "ACME",
    "shares": 100
}
>>>
```

Если вы хотите, чтобы при выводе происходила сортировка ключей, используйте аргумент *sort_keys*:

```
>>> print(json.dumps(data, sort_keys=True))
{"name": "ACME", "price": 542.23, "shares": 100}
>>>
```

Экземпляры в обычном случае не являются сериализуемыми. Например:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p = Point(2, 3)
>>> json.dumps(p)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/json/__init__.py", line 226, in dumps
    return _default_encoder.encode(obj)
  File "/usr/local/lib/python3.3/json/encoder.py", line 187, in encode
    chunks = self.iterencode(o, _one_shot=True)
  File "/usr/local/lib/python3.3/json/encoder.py", line 245, in iterencode
    return _iterencode(o, 0)
  File "/usr/local/lib/python3.3/json/encoder.py", line 169, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: <__main__.Point object at 0x1006f2650> is not JSON serializable
>>>
```

Если вы хотите сериализовать экземпляры, вы можете предоставить функцию, которая принимает экземпляр на вход и возвращает словарь, который может быть сериализован. Например:

```
def serialize_instance(obj):
    d = { '__classname__' : type(obj).__name__ }
    d.update(vars(obj))
    return d
```

Если вы хотите получить экземпляр обратно, вы можете сделать это так:

```
# Словарь отображения имён на известные классы
classes = {
    'Point' : Point
}

def unserialize_object(d):
    clsname = d.pop('__classname__', None)
    if clsname:
        cls = classes[clsname]
        obj = cls.__new__(cls) # Создание экземпляра без вызова __init__
        for key, value in d.items():
            setattr(obj, key, value)
        return obj
    else:
        return d
```

Вот пример того, как используются эти функции:

```
>>> p = Point(2,3)
>>> s = json.dumps(p, default=serialize_instance)
>>> s
'{"__classname__": "Point", "y": 3, "x": 2}'
>>> a = json.loads(s, object_hook=unserialize_object)
>>> a
<__main__.Point object at 0x1017577d0>
>>> a.x
2
>>> a.y
3
>>>
```

В модуле *json* множество других возможностей для контроля низкоуровневой интерпретации чисел, специальных значений (таких как NaN) и т.п. Обратитесь к [документации](#) за подробностями.

6.3. Парсинг простых XML-данных

Задача

Вы хотите извлечь данные из простого XML-документа.

Решение

Модуль `xml.etree.ElementTree` может быть использован для извлечения данных из простых XML-документов. Чтобы продемонстрировать это, предположим, что вы хотите распарсить и подготовить выжимку RSS-фида [Planet Python](#). Вот скрипт, который это сделает:

```
from urllib.request import urlopen
from xml.etree.ElementTree import parse

# Скачивание и парсинг RSS-фида
u = urlopen('http://planet.python.org/rss20.xml')
doc = parse(u)

# Извлечение и вывод нужных тегов
for item in doc.iterfind('channel/item'):
    title = item.findtext('title')
    date = item.findtext('pubDate')
    link = item.findtext('link')

    print(title)
    print(date)
    print(link)
    print()
```

Если вы запустите вышеприведённый скрипт, вывод будет примерно таким:

```
Steve Holden: Python for Data Analysis
Mon, 19 Nov 2012 02:13:51 +0000
http://holdenweb.blogspot.com/2012/11/python-for-data-analysis.html
```

```
Vasudev Ram: The Python Data model (for v2 and v3)
Sun, 18 Nov 2012 22:06:47 +0000
http://jugad2.blogspot.com/2012/11/the-python-data-model.html
```

```
Python Diary: Been playing around with Object Databases
Sun, 18 Nov 2012 20:40:29 +0000
http://www.pythondiary.com/blog/Nov.18,2012/been----object-databases.ht
```

```
Vasudev Ram: Wakari, Scientific Python in the cloud
Sun, 18 Nov 2012 20:19:41 +0000
http://jugad2.blogspot.com/2012/11/wakari-scientific-python-in-cloud.htm
```

```
Jesse Jiryu Davis: Toro: synchronization primitives for Tornado coroutines
Sun, 18 Nov 2012 20:17:49 +0000
http://feedproxy.google.com/~r/EmptysquarePython/~3/_DOZT2Kd0hQ/
```

Очевидно, что если вы хотите провести дополнительную обработку, вам нужно заменить инструкции *print()* на что-то более интересное.

Обсуждение

В очень многих приложениях нужно работать с XML-данными. XML не только широко используется в качестве формата для обмена данными через интернет, это также распространенный формат хранения данных приложений (обработка текста, музыкальные библиотеки и т.п.)
Нижеследующее обсуждение подразумевает, что читатель уже знаком с основами XML.

Во многих случаях, когда XML просто используется для хранения данных, структура документа проста и прямолинейна. Например, RSS-поток из примера выглядит примерно так:

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
<channel>
    <title>Planet Python</title>
    <link>http://planet.python.org/</link>
    <language>en</language>
    <description>Planet Python – http://planet.python.org/</description>
    <item>
        <title>Steve Holden: Python for Data Analysis</title>
        <guid>http://holdenweb.blogspot.com/...-data-analysis.html</guid>
        <link>http://holdenweb.blogspot.com/...-data-analysis.html</link>
        <description>...</description>
        <pubDate>Mon, 19 Nov 2012 02:13:51 +0000</pubDate>
    </item>
    <item>
        <title>Vasudev Ram: The Python Data model (for v2 and v3)</title>
        <guid>http://jugad2.blogspot.com/...-data-model.html</guid>
        <link>http://jugad2.blogspot.com/...-data-model.html</link>
        <description>...</description>
        <pubDate>Sun, 18 Nov 2012 22:06:47 +0000</pubDate>
    </item>
```

```
<item>
    <title>Python Diary: Been playing around with Object Databases</title>
    <guid>http://www.pythondiary.com/...-object-databases.html</guid>
    <link>http://www.pythondiary.com/...-object-databases.html</link>
    <description>...</description>
    <pubDate>Sun, 18 Nov 2012 20:40:29 +0000</pubDate>
</item>
...
</channel>
</rss>
```

Функция `xml.etree.ElementTree.parse()` парсит весь XML-документ в объект `document`. Далее вы можете использовать такие методы, как `find()`, `iterfind()` и `findtext()`, для поиска определённых XML-элементов. Аргументы этих функций — это имена определенных тегов, такие как `channel/item` или `title`.

Когда вы задаёте теги, вы должны принимать во внимание всю структуру документа. Каждая операция поиска предпринимается относительно стартового элемента. Тег, который вы предоставляеме каждой операции, также рассматривается относительно старта. В вышеприведённом примере вызов `doc.iterfind('channel/item')` найдёт все элементы “item” под элементом “channel”. `doc` представляет вершину документа (высший уровень — элемент “rss”). Последующие вызовы `item.findtext()` будут делаться относительно найденных элементов “item”.

Каждый элемент, представленный модулем `ElementTree`, имеет несколько основных атрибутов и методов, весьма полезных при парсинге. Атрибут `tag` содержит имя тега, атрибут `text` содержит замкнутый текст, а метод `get()` может быть использован для извлечения атрибутов (если они присутствуют). Например:

```
>>> doc
<xml.etree.ElementTree.ElementTree object at 0x101339510>
>>> e = doc.find('channel/title')
>>> e
<Element 'title' at 0x10135b310>
>>> e.tag
'title'
>>> e.text
'Planet Python'
>>> e.get('some_attribute')
>>>
```

Стоит отметить, что `xml.etree.ElementTree` — не единственный способ

парсинга XML. Для более продвинутых приложений вы можете попробовать `lxml`. Эта библиотека использует тот же интерфейс, что и `ElementTree`, так что вышеупомянутые примеры будут работать так же. Вы просто должны изменить первую инструкцию `import` на `from lxml.etree import parse`.

Библиотека `lxml` имеет преимущество — полное соответствие стандартам XML. Она также чрезвычайно быстро работает и предоставляет поддержку таких возможностей, как валидация, XSLT и XPath.

6.4. Инкрементальный парсинг очень больших XML-файлов

Задача

Вам нужно извлечь данные из огромного XML-документа, используя как можно меньше памяти.

Решение

Каждый раз, когда вы сталкиваетесь с инкрементальной обработкой данных, вы должны вспоминать об итераторах и генераторах. Вот простая функция, которая может быть использована для инкрементальной обработки огромных XML-файлов при очень небольшом потреблении памяти:

```
from xml.etree.ElementTree import iterparse

def parse_and_remove(filename, path):
    path_parts = path.split('/')
    doc = iterparse(filename, ('start', 'end'))
    # Пропуск корневого элемента
    next(doc)

    tag_stack = []
    elem_stack = []
    for event, elem in doc:
        if event == 'start':
            tag_stack.append(elem.tag)
            elem_stack.append(elem)
        elif event == 'end':
            if tag_stack == path_parts:
                yield elem
                elem_stack[-2].remove(elem)
    try:
```

```
    tag_stack.pop()
    elem_stack.pop()
except IndexError:
    pass
```

Чтобы протестировать функцию, вам потребуется большой XML-файл. Часто такие файлы можно найти на государственных сайтах и ресурсах с открытой информацией. Например, вы можете скачать [базу данных выбоин на дорогах Чикаго](#) в формате XML. Когда писалась эта книга, этот файл состоял из более чем 100 000 строк данных, которые были закодированы так:

```
<response>
<row>
    <row ...>
        <creation_date>2012-11-18T00:00:00</creation_date>
        <status>Completed</status>
        <completion_date>2012-11-18T00:00:00</completion_date>
        <service_request_number>12-01906549</service_request_number>
        <type_of_service_request>Pot Hole in Street</type_of_service_request>
        <current_activity>Final Outcome</current_activity>
        <most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
        <street_address>4714 S TALMAN AVE</street_address>
        <zip>60632</zip>
        <x_coordinate>1159494.68618856</x_coordinate>
        <y_coordinate>1873313.83503384</y_coordinate>
        <ward>14</ward>
        <police_district>9</police_district>
        <community_area>58</community_area>
        <latitude>41.808090232127896</latitude>
        <longitude>-87.69053684711305</longitude>
        <location latitude="41.808090232127896"
                  longitude="-87.69053684711305" />
    /row>
    <row ...>
        <creation_date>2012-11-18T00:00:00</creation_date>
        <status>Completed</status>
        <completion_date>2012-11-18T00:00:00</completion_date>
        <service_request_number>12-01906695</service_request_number>
        <type_of_service_request>Pot Hole in Street</type_of_service_request>
        <current_activity>Final Outcome</current_activity>
        <most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
        <street_address>3510 W NORTH AVE</street_address>
        <zip>60647</zip>
        <x_coordinate>1152732.14127696</x_coordinate>
        <y_coordinate>1910409.38979075</y_coordinate>
        <ward>26</ward>
        <police_district>14</police_district>
        <community_area>23</community_area>
        <latitude>41.91002084292946</latitude>
```

```
<longitude>-87.71435952353961</longitude>
<location latitude="41.91002084292946"
          longitude="-87.71435952353961" />
</row>
</row>
</response>
```

Предположим, что вы хотите написать скрипт, который отсортирует ZIP-коды по количеству отчётов о выбоинах. Чтобы сделать это, вы можете написать такой код:

```
from xml.etree.ElementTree import parse
from collections import Counter

potholes_by_zip = Counter()

doc = parse('potholes.xml')
for pothole in doc.iterfind('row/row'):
    potholes_by_zip[pothole.findtext('zip')] += 1
for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

Единственная проблема с этим скриптом заключается в том, что он читает в память XML-файл целиком. На нашем компьютере при запуске он отъел 450 мегабайт оперативной памяти. Если же применить код из этого рецепта, программа изменится совсем чуть-чуть:

```
from collections import Counter
potholes_by_zip = Counter()

data = parse_and_remove('potholes.xml', 'row/row')
for pothole in data:
    potholes_by_zip[pothole.findtext('zip')] += 1

for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)
```

Но эта версия занимает при запуске всего 7 мегабайт оперативной памяти — огромная экономия налицо!

Обсуждение

Этот рецепт основывается на двух базовых возможностях модуля *ElementTree*. Метод *iterparse()* позволяет обрабатывать XML-документы

инкрементально. Чтобы использовать его, вы передаёте имя файла вместе со списком событий, состоящим из одного или более следующих аргументов: *start*, *end*, *start-ns* и *end-ns*. Итератор, созданный *iterparse()*, производит кортежи формата (*event*, *elem*), где *event* — одно из событий списка, а *elem* — полученный XML-элемент. Например:

```
>>> data = iterparse('potholes.xml', ('start', 'end'))
>>> next(data)
('start', <Element 'response' at 0x100771d60>)
>>> next(data)
('start', <Element 'row' at 0x100771e68>)
>>> next(data)
('start', <Element 'row' at 0x100771fc8>)
>>> next(data)
('start', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('end', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('start', <Element 'status' at 0x1006a7f18>)
>>> next(data)
('end', <Element 'status' at 0x1006a7f18>)
>>>
```

События *start* создаются, когда элемент создан, но еще не наполнен любыми другими данными (например, элементами-потомками). События *end* создаются, когда элемент завершен. Хотя в данном рецепте это и не показано, события *start-ns* и *end-ns* используются для работы с объявлениями пространств имён XML.

В этом рецепте события *start* и *end* используются для управления стеками элементов и тегов. Стеки представляют текущую иерархическую структуру документа в процессе его парсинга, а также используются для определения того, совпадает ли элемент с запрашиваемым путём, переданным в функцию *parse_and_remove()*. Если совпадение произошло, *yield* выдаёт его обратно вызывавшему.

Следующая инструкция после *yield* — базовая возможность *Element.Tree*, которая позволяет этому рецепту экономить память:

```
elem_stack[-2].remove(elem)
```

Эта инструкция удаляет выданный ранее элемент из его родителя. Исходя из предположения, что нигде более на него не осталось ссылок, элемент

уничтожается, а память высвобождается.

Конечный эффект итеративного парсинга и удаления узлов — крайне эффективный инкрементальный проход по документу. Ни на одном этапе не создается полное дерево документа. Однако можно написать код, который обрабатывает XML-данные прямолинейным способом.

Главный недостаток этого рецепта — производительность. При тестировании версия, которая читает весь документ в память, отработала в 2 раза быстрее, чем инкрементальная. Однако она потребовала в 60 раз больше памяти. Так что если память важна, инкрементальный подход даёт большой выигрыш.

6.5. Преобразование словарей в XML

Задача

Вы хотите взять данные из словаря Python и превратить их в XML.

Решение

Хотя библиотека `xml.etree.ElementTree` обычно используется для парсинга, её также можно применить для создания XML-документов. Например, посмотрите на такую функцию:

```
from xml.etree.ElementTree import Element

def dict_to_xml(tag, d):
    """
    Превращает простой словарь пар ключ/значение в XML
    """
    elem = Element(tag)
    for key, val in d.items():
        child = Element(key)
        child.text = str(val)
        elem.append(child)
    return elem
```

Вот пример её работы:

```
>>> s = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
>>> e = dict_to_xml('stock', s)
>>> e
```

```
<Element 'stock' at 0x1004b64c8>
>>>
```

Результатом этого преобразования является экземпляр *Element*. Для ввода-вывода его можно легко конвертировать в байтовую строку — для этого нужно использовать функцию *tostring()* из модуля *xml.etree.ElementTree*.

Например:

```
>>> from xml.etree.ElementTree import tostring
>>> tostring(e)
b'<stock><price>490.1</price><shares>100</shares><name>GOOG</name></stoc
>>>
```

Если вы хотите прикрепить атрибуты к элементу, используйте метод *set()*:

```
>>> e.set('_id','1234')
>>> tostring(e)
b'<stock _id="1234"><price>490.1</price><shares>100</shares><name>GOOG<,
</stock>'
>>>
```

Если порядок элементов имеет значение, подумайте над созданием *OrderedDict* вместо обычного словаря (см. [рецепт 1.7.](#))

Обсуждение

При генерации XML вы можете склоняться к простому созданию строк.

Например:

```
def dict_to_xml_str(tag, d):
    """
    Превращает простой словарь пар ключ/значение в XML
    """

    parts = ['<{}>'.format(tag)]
    for key, val in d.items():
        parts.append('<{}>{}</{}>'.format(key, val))
        parts.append('</{}>'.format(tag))
    return ''.join(parts)
```

Проблема в том, что вы влезете в большие неприятности, если попытаетесь сделать это вручную. Например, что случится, если значения словаря будут содержать спецсимволы? Например:

```
>>> d = { 'name' : '<spam>' }

>>> # Создание строки
>>> dict_to_xml_str('item',d)
'<item><name><spam></name></item>'

>>> # Правильное создание XML
>>> e = dict_to_xml('item',d)
>>> tostring(e)
b'<item><name>&lt;spam&gt;</name></item>'
>>>
```

Обратите внимание, как в последнем примере символы < и > заменяются на < и >.

Для справки: если вам когда-либо потребуется вручную экранировать или деэкранировать такие символы, вы можете использовать функции `escape()` и `unescape()` из модуля `xml.sax.saxutils`. Например:

```
>>> from xml.sax.saxutils import escape, unescape
>>> escape('<spam>')
'&lt;spam&gt;'
>>> unescape(_)
'<spam>'
>>>
```

Если оставить в стороне создание правильного вывода, другая причина создавать экземпляры `Element` вместо строк заключается в том, что их легче объединять друг с другом для создания более крупного документа. Получающиеся экземпляры `Element` также могут быть обработаны различными способами без необходимости парсить XML-текст. Вы можете провести всю обработку данных высокоуровневым способом, а затем в самом конце вывести их в строковой форме.

6.6. Парсинг, изменение и перезапись XML

Задача

Вы хотите прочесть XML-документ, изменить его, а затем записать обратно в форме XML.

Решение

Модуль `xml.etree.ElementTree` облегчает выполнение таких задач. Вы можете начать с парсинга документа обычным способом. Предположим, например, что у вас есть документ под названием `pred.xml`, который выглядит так:

```
<?xml version="1.0"?>
<stop>
  <id>14791</id>
  <nm>Clark & Balmoral</nm>
  <sri>
    <rt>22</rt>
    <d>North Bound</d>
    <dd>North Bound</dd>
  </sri>
  <cr>22</cr>
  <pre>
    <pt>5 MIN</pt>
    <fd>Howard</fd>
    <v>1378</v>
    <rn>22</rn>
  </pre>
  <pre>
    <pt>15 MIN</pt>
    <fd>Howard</fd>
    <v>1867</v>
    <rn>22</rn>
  </pre>
</stop>
```

Вот пример того, как можно использовать `ElementTree` для прочтения и документа и изменения его структуры:

```
>>> from xml.etree.ElementTree import parse, Element
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>

>>> # Удалим несколько элементов
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))

# Вставка нового элемента после <nm>...</nm>
>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
```

```
>>> e.text = 'This is a test'  
>>> root.insert(2, e)  
  
>>> # Запись обратно в файл  
>>> doc.write('newpred.xml', xml_declaration=True)  
>>>
```

Результатом этих операций будет новый XML-файл, который выглядит так:

```
<?xml version='1.0' encoding='us-ascii'?>  
<stop>  
  <id>14791</id>  
  <nm>Clark & Balmoral</nm>  
  <spam>This is a test</spam>  
  <pre>  
    <pt>5 MIN</pt>  
    <fd>Howard</fd>  
    <v>1378</v>  
    <rn>22</rn>  
  </pre>  
  <pre>  
    <pt>15 MIN</pt>  
    <fd>Howard</fd>  
    <v>1867</v>  
    <rn>22</rn>  
  </pre>  
</stop>
```

Обсуждение

Изменение структуры XML-документа — незамысловатый процесс, но вы должны помнить, что все изменения в общем применяются к родительскому элементу, и они обращаются с ним как со списком. Например, если вы уберёте элемент, он будет убран из его непосредственного родителя путём использования метода `remove()` родителя. Если вы вставляете или добавляете новые элементы в конец, вы также применяете методы `insert()` и `append()` к родителю. Также можно манипулировать элементами с помощью операций индексирования и извлечения среза, таких как `element[i]` или `element[i:j]`.

Если вы хотите создать новые элементы, используйте класс `Element`, как показано в этом рецепте. Это также описано в [рецепте 6.5](#).

6.7. Парсинг XML-документов с

пространствами имён

Задача

Вам нужно распарсить XML-документ, но он использует пространства имён XML.

Решение

Предположим, что у нас есть документ, использующий пространства имён:

```
<?xml version="1.0" encoding="utf-8"?>
<top>
    <author>David Beazley</author>
    <content>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>Hello World</title>
            </head>
            <body>
                <h1>Hello World!</h1>
            </body>
        </html>
    </content>
</top>
```

Если вы парсите этот документ и пытаетесь выполнить обычные запросы, вы обнаружите, что это не работает так просто, поскольку всё становится невероятно многословным:

```
>>> # Некоторые запросы, которые работают
>>> doc.findtext('author')
'David Beazley'
>>> doc.find('content')
<Element 'content' at 0x100776ec0>

>>> # Запрос с использованием пространства имён (не работает)
>>> doc.find('content/xhtml')

>>> # Работает при полном определении
>>> doc.find('content/{http://www.w3.org/1999/xhtml}html')
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>

>>> # Не работает
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/head/title')
```

```
>>> # Полностью определён
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/'
... '{http://www.w3.org/1999/xhtml}head/{http://www.w3.org/1999/xhtml}title')
'Hello World'
>>>
```

Часто вы можете упростить дело путем заворачивания работы с пространством имён во вспомогательный класс:

```
class XMLNamespaces:
    def __init__(self, **kwargs):
        self.namespaces = {}
        for name, uri in kwargs.items():
            self.register(name, uri)
    def register(self, name, uri):
        self.namespaces[name] = '{'+uri+'}'
    def __call__(self, path):
        return path.format_map(self.namespaces)
```

Чтобы использовать этот класс, вы можете поступить так:

```
>>> ns = XMLNamespaces(html='http://www.w3.org/1999/xhtml')
>>> doc.find(ns('content/{html}html'))
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> doc.findtext(ns('content/{html}html/{html}head/{html}title'))
'Hello World'
>>>
```

Обсуждение

Парсинг XML-документов, содержащих пространства имён, может быть запутанным. Класс `XMLNamespaces` на самом деле предназначен для облегчения этой задачи: он позволяет использовать сокращённые имена для пространств имён в последующих операциях, а не полные URI.

К несчастью, в базовом парсере `ElementTree` нет механизма для получения дополнительной информации о пространствах имён. Однако вы можете получить немного больше информации об области видимости обработки пространств имён, если будете использовать функцию `iterparse()`. Например:

```
>>> from xml.etree.ElementTree import iterparse
>>> for evt, elem in iterparse('ns2.xml', ('end', 'start-ns', 'end-ns')):
...     print(evt, elem)
```

```
...
end <Element 'author' at 0x10110de10>
start-ns ('', 'http://www.w3.org/1999/xhtml')
end <Element '{http://www.w3.org/1999/xhtml}title' at 0x1011131b0>
end <Element '{http://www.w3.org/1999/xhtml}head' at 0x1011130a8>
end <Element '{http://www.w3.org/1999/xhtml}h1' at 0x101113310>
end <Element '{http://www.w3.org/1999/xhtml}body' at 0x101113260>
end <Element '{http://www.w3.org/1999/xhtml}html' at 0x10110df70>
end-ns None
end <Element 'content' at 0x10110de68>
end <Element 'top' at 0x10110dd60>
# Это самый верхний элемент
>>> elem
<Element 'top' at 0x10110dd60>
>>>
```

Последнее замечание: если текст, который вы парсите, использует пространства имён в дополнение к другим продвинутым возможностям XML, вам лучше перейти с *ElementTree* на библиотеку *lxml*. Например, она предоставляет улучшенную поддержку валидации документов по DTD, более полную поддержку XPath и другие продвинутые возможности. А этот рецепт — просто небольшой фикс для облегчения парсинга.

6.8. Взаимодействие с реляционной базой данных

Задача

Вам нужно выбирать, вставлять или удалять строки (*rows*) в реляционной базе данных.

Решение

Стандартный способ представления строк (*rows*) данных в Python — это последовательность кортежей. Например:

```
stocks = [
    ('GOOG', 100, 490.1),
    ('AAPL', 50, 545.75),
    ('FB', 150, 7.45),
    ('HPQ', 75, 33.2),
]
```

Если данные представлены в такой форме, относительно легко наладить взаимодействие с реляционной базой данных, используя стандартный API баз данных Python, как описано в [PEP 249](#). Суть API в том, что все операции с базой данных выполняются с помощью SQL-запросов. Каждая строка вводимых или выводимых данных представлена в форме кортежа.

Чтобы попробовать это в деле, вы можете воспользоваться модулем `sqlite3`, который входит в стандартную поставку Python. Если вы используете другую базу данных (например, MySQL, Postgres, ODBC), вы должны будете установить стороннюю библиотеку. Однако программный интерфейс будет практически таким же, если не идентичным.

Первый шаг — подсоединиться к базе данных. Обычно для этого нужно вызвать функцию `connect()` и передать ей такие параметры, как имя базы данных, имя хоста, имя пользователя, пароль и т.п. Например:

```
>>> import sqlite3  
>>> db = sqlite3.connect('database.db')  
>>>
```

Чтобы что-то делать с данными, нужно создать курсор. Когда у вас есть курсор, вы можете выполнять SQL-запросы. Например:

```
>>> c = db.cursor()  
>>> c.execute('create table portfolio (symbol text, shares integer, price real)')  
<sqlite3.Cursor object at 0x10067a730>  
>>> db.commit()  
>>>
```

Чтобы вставить последовательность строк в данные, используйте такую инструкцию:

```
>>> c.executemany('insert into portfolio values (?,?,?,?)', stocks)  
<sqlite3.Cursor object at 0x10067a730>  
>>> db.commit()  
>>>
```

Чтобы сделать запрос, используйте такую инструкцию:

```
>>> for row in db.execute('select * from portfolio'):  
...     print(row)  
... 
```

```
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
('FB', 150, 7.45)
('HPQ', 75, 33.2)
>>>
```

Если вы хотите сделать запросы, которые принимают поставляемые пользователем входные параметры, убедитесь, что вы экранируете параметры, используя символ ?:

```
>>> min_price = 100
>>> for row in db.execute('select * from portfolio where price >= ?',
                           (min_price,)):
    ...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
>>>
```

Обсуждение

На низком уровне взаимодействие с базой данных выполняется абсолютно прямолинейно. Вы просто формируете SQL-запросы и скармливаете их модулю, чтобы либо обновить информацию в базе, либо извлечь данные. Тем не менее, есть тонкие моменты, с которыми в некоторых случаях придётся разбираться.

Одно из возможных осложнений — отображение данных из базы на типы Python. Для записей типа дат наиболее частым случаем будет использование экземпляров *datetime* из одноимённого модуля, или системных таймстемпов с применением модуля *time*. Для числовых данных, и особенно финансовых данных, в которых применяются десятичные дроби, может применяться представление чисел как экземпляров *Decimal* из модуля *decimal*. К сожалению, конкретные принципы отображения варьируются в зависимости от бэкенда базы данных, так что вам придется почитать документацию.

Ещё одно критически важное осложнение касается формирования строк с инструкциями SQL. Вы никогда не должны использовать операторы форматирования строк Python (например, %) или метод *.format()* для создания таких строк. Если значения, предоставленные таким операторам форматирования, вводятся пользователями, это открывает вашу программу для SQL-инъекций (см. <http://xkcd.com/327>). Специальный подменяющий

символ ? в запросах требует от бэкенда базы данных использовать его собственный механизм подстановки строк, который (будем надеяться) делает это безопасно.

К сожалению, существует некоторое разнообразие в том, как бэкенды различных баз данных интерпретируют символы подстановки. Многие модули используют ? или %s, тогда как другие могут использовать другой символ, такой как :0 или :1, чтобы ссылаться на параметры. Вам нужно обратиться к документации используемого модуля базы данных. Атрибут *paramstyle* модуля базы данных также содержит информацию о стиле использования кавычек.

Для простого взаимодействия с таблицей базы данных использовать API обычно очень просто. Если вы делаете что-то более нетривиальное, имеет смысл использовать высокоуровневый интерфейс, такой как объектно-реляционные отображатели (ORM). Библиотеки типа [SQLAlchemy](#) позволяют описывать таблицы базы данных как классы Python, и выполнять операции с базами данных, скрывая весь лежащий в основе SQL.

6.9. Декодирование и кодирование шестнадцатеричных цифр

Задача

Вам нужно декодировать строку шестнадцатеричных цифр в байтовую строку или закодировать байтовую строку в шестнадцатеричное представление.

Решение

Если вам просто нужно декодировать или закодировать сырую строку шестнадцатеричных цифр, используйте модуль *binascii*. Например:

```
>>> # Изначальная байтовая строка
>>> s = b'hello'

>>> # Закодировать в hex
>>> import binascii
>>> h = binascii.b2a_hex(s)
>>> h
b'68656c6c6f'
```

```
>>> # Декодировать обратно в байты
>>> binascii.a2b_hex(h)
b'hello'
>>>
```

Похожую функцию можно найти в модуле *base64*. Например:

```
>>> import base64
>>> h = base64.b16encode(s)
>>> h
b'68656C6C6F'
>>> base64.b16decode(h)
b'hello'
>>>
```

Обсуждение

По большей части конвертирование в и из шестнадцатеричную форму с помощью показанных приёмов не составляет труда. Главная разница между этими двумя техниками заключается в приведении к регистру. Функции *base64.b16decode()* и *base64.b16encode()* работают только с шестнадцатеричными символами в верхнем регистре, а функции из модуля *binascii* могут работать с обоими регистрами.

Также важно отметить, что вывод, который производят кодирующие функции, всегда является байтовой строкой. Чтобы принудительно вывести его в Unicode, вам придется добавить дополнительный шаг. Например:

```
>>> h = base64.b16encode(s)
>>> print(h)
b'68656C6C6F'
>>> print(h.decode('ascii'))
68656C6C6F
>>>
```

При декодировании шестнадцатеричных цифр функции *16decode()* и *a2b_hex()* принимают и байтовые, и юникодовые строки. Однако эти строки должны содержать только закодированные в ASCII шестнадцатеричные цифры.

6.10. Кодирование и декодирование в Base64

Задача

Вам нужно декодировать или закодировать бинарные данные, используя кодировку Base64.

Решение

В модуле `base64` есть две функции, которые делают именно то, что вам нужно: `b64encode()` и `b64decode()`. Например:

```
>>> # Какие-то байтовые данные
>>> s = b'hello'
>>> import base64

>>> # Закодировать в Base64
>>> a = base64.b64encode(s)
>>> a
b'aGVsbG8='

>>> # Декодировать из Base64
>>> base64.b64decode(a)
b'hello'
>>>
```

Обсуждение

Кодировка Base64 предназначена только для использования с байт-ориентированными данными, такими как байтовые строки и байтовые массивы. Более того, вывод процесса кодирования всегда будет байтовой строкой. Если вы смешиваете данные в Base64 с текстом в Unicode, вам придется выполнить дополнительный шаг для декодирования. Например:

```
>>> a = base64.b64encode(s).decode('ascii')
>>> a
'aGVsbG8='
>>>
```

При декодировании Base64 могут быть предоставлены и байтовые строки, и текстовые строки в Unicode. Однако строки Unicode могут содержать только символы ASCII.

6.11. Чтение и запись бинарных массивов

структур

Задача

Вы хотите прочесть или записать данные, закодированные как бинарный массив из единообразных структур, в кортежи Python.

Решение

Чтобы работать с бинарными данными, используйте модуль *struct*. Вот пример кода, который записывает список кортежей Python в бинарный файл, кодируя каждый кортеж в структуру с помощью модуля *struct*:

```
from struct import Struct

def write_records(records, format, f):
    """
    Записывает последовательность кортежей в бинарный файл структур.
    """

    record_struct = Struct(format)
    for r in records:
        f.write(record_struct.pack(*r))

# Пример
if __name__ == '__main__':
    records = [ (1, 2.3, 4.5),
                (6, 7.8, 9.0),
                (12, 13.4, 56.7) ]

    with open('data.b', 'wb') as f:
        write_records(records, '<idd', f)
```

Есть несколько подходов к обратному превращению этого файла в список кортежей. Во-первых, если вы читаете файл кусочками (чанками) инкрементально, вы можете написать такой код:

```
from struct import Struct

def read_records(format, f):
    record_struct = Struct(format)
    chunks = iter(lambda: f.read(record_struct.size), b'')
    return (record_struct.unpack(chunk) for chunk in chunks)

# Пример
```

```
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        for rec in read_records('<idd', f):
            # Обработка записи
            ...

```

Если вы хотите прочесть файл целиком в байтовую строку за один проход и преобразовывать его кусочек за кусочком, вы можете сделать это так:

```
from struct import Struct

def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack_from(data, offset)
            for offset in range(0, len(data), record_struct.size))

# Пример
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        data = f.read()
        for rec in unpack_records('<idd', data):
            # Обработка записи
            ...

```

В обоих случаях результатом будет итерируемый объект, который производит кортежи, которые были сохранены в файле при его создании.

Обсуждение

В программах, которые должны кодировать и декодировать бинарные данные, обычно используют модуль *struct*. Чтобы объявить новую структуру, просто создайте экземпляр *Struct*, как показано ниже:

```
# 32-битное целое число (little endian), два числа с плавающей точкой
# двойной точности
record_struct = Struct('<idd')
```

Структуры всегда определяются путём использования набора кодов структур, таких как *i*, *d*, *f* и так далее (см. документацию [Python](#)). Эти коды соответствуют определенным бинарным типам данных, таким как 32-битные целые числа, 64-битные числа с плавающей точкой, 32-битные числа с плавающей точкой и т.д. Символ < в качестве первого символа определяет порядок следования байтов. В этом примере он задает порядок little endian. Замените символ на >, чтобы задать big endian, или на ! для сетевого порядка

байтов.

Полученный экземпляр Struct имеет различные атрибуты и методы для манипулирования структурами этого типа. Атрибут `size` содержит размер структуры в байтах, что полезно для операций ввода-вывода. Методы `pack()` и `unpack()` используются для упаковки и распаковки данных. Например:

Иногда вы можете увидеть, что операции `pack()` и `unpack()` вызываются, как функции уровня модуля:

```
>>> import struct  
>>> struct.pack('<idd', 1, 2.0, 3.0)  
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x00\x00'  
>>> struct.unpack('<idd', _)  
(1, 2.0, 3.0)  
>>>
```

Это работает, но менее элегантно, нежели создание единственного экземпляра *Struct* – особенно если одна и та же структура появляется во многих местах вашего кода. Путём создания экземпляра *Struct* форматирующий код определяется только единожды, и все полезные операции прекрасным образом сгруппированы вместе. Это точно увеличит легкость поддерживания вашего кода, ведь вам придется вносить изменения только в одном месте.

Код для чтения бинарных структур использует несколько интересных и элегантных идиом программирования. В функции `read_records()` функция `iter()` используется для создания итератора, который возвращает кусочки (чанки) фиксированного размера (см. [рецепт 5.8.](#)) Этот итератор раз за разом вызывает переданный пользователем вызываемый объект (в данном случае `lambda: f.read(record_struct.size)`), пока он не вернёт определённое значение (в данном случае `b`), на чём итерации останавливаются. Например:

```
>>> f = open('data.b', 'rb')
>>> chunks = iter(lambda: f.read(20), b'')
>>> chunks
<callable_iterator object at 0x10069e6d0>
>>> for chk in chunks:
...     print(chk)
...
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x12@'
b'\x06\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"@"
b'\x0c\x00\x00\x00\xcd\xcc\xcc\xcc\xcc\xcc*x@\x9a\x99\x99\x99\x99YL@'
>>>
```

Смысл использования итератора в том, что он позволяет записям создаваться с помощью генератора (generator comprehension), как показано в примере. Если бы вы не использовали это решение, ваш код мог бы выглядеть так:

```
def read_records(format, f):
    record_struct = Struct(format)
    while True:
        chk = f.read(record_struct.size)
        if chk == b'':
            break
        yield record_struct.unpack(chk)
return records
```

В функции *unpack_records()* используется другой подход — метод *unpack_from()*. Это полезный метод для извлечения бинарных данных из более крупного бинарного массива, потому что он делает это без создания временных объектов или копий в памяти. Вы просто передаете ему байтовую строку (или любой массив) вместе с байтовым сдвигом (*offset*), и он распакует поля прямо из этого места.

Если вы использовали *unpack()* вместо *unpack_from()*, вы можете захотеть изменить код, чтобы создать большое количество маленьких срезов и вычислений сдвига. Например:

```
def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack(data[offset:offset
        + record_struct.size])
        for offset in range(0, len(data), record_struct.size))
```

В дополнение к тому, что эта версия сложнее читается, она также требует

намного больше работы, поскольку она выполняет различные вычисления сдвига, копирует данные и создаёт объекты среза. Если вы будете распаковывать много структур из большой байтовой строки, которую уже прочитали, *unpack_from()* будет более элегантным решением.

Распаковка записей — одна из областей, где могут найти применение объекты *namedtuple* из модуля *collections*. Они позволят вам установить имена атрибутов на возвращаемые кортежи. Например:

```
from collections import namedtuple

Record = namedtuple('Record', ['kind', 'x', 'y'])

with open('data.p', 'rb') as f:
    records = (Record(*r) for r in read_records('<idd', f))

for r in records:
    print(r.kind, r.x, r.y)
```

Если вы пишете программу, которой нужно работать с большим количеством бинарных данных, вам стоит использовать библиотеку типа *pintpy*. Например, вместо чтения бинарного файла в список кортежей вы можете прочесть его в структурированный массив:

```
>>> import numpy as np
>>> f = open('data.b', 'rb')
>>> records = np.fromfile(f, dtype='<i,<d,<d')
>>> records
array([(1, 2.3, 4.5), (6, 7.8, 9.0), (12, 13.4, 56.7)],
      dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '<f8')])
>>> records[0]
(1, 2.3, 4.5)
>>> records[1]
(6, 7.8, 9.0)
>>>
```

И последнее: если вы столкнулись с задачей чтения бинарных данных в каком-то известном формате (например, форматах растровых или векторных изображений, HDF5 и т.д.), проверьте, нет ли в Python модуля для работы с ними. Не стоит изобретать велосипед, если можно обойтись без этого.

6.12. Чтение вложенных и различных по размеру бинарных структур

Задача

Вам нужно прочесть сложные бинарные данные, которые содержат коллекцию вложенных записей и/или записей различного размера. Такие данные могут включать изображения, видео, векторные изображения и т.д.

Решение

Модуль *struct* может быть использован для декодирования и кодирования бинарных данных практически любой структуры. Чтобы проиллюстрировать работу с задачей этого рецепта, предположим, что у вас есть структура данных Python, представляющая точки, составляющие набор многоугольников:

```
polys = [
    [ (1.0, 2.5), (3.5, 4.0), (2.5, 1.5) ],
    [ (7.0, 1.2), (5.1, 3.0), (0.5, 7.5), (0.8, 9.0) ],
    [ (3.4, 6.3), (1.2, 0.5), (4.6, 9.2) ],
]
```

Теперь предположим, что данные были закодированы в бинарный файл, который начинается следующим заголовком:

Byte	Type	Description
0	int	File code (0x1234, little endian)
4	double	Minimum x (little endian)
12	double	Minimum y (little endian)
20	double	Maximum x (little endian)
28	double	Maximum y (little endian)
36	int	Number of polygons (little endian)

После заголовка идёт набор многоугольников, каждый из которых закодирован так:

Byte	Type	Description
0	int	Record length including length (N bytes)
4-N	Points	Pairs of (X,Y) coords as doubles

Чтобы записать этот файл, вы можете использовать такой код:

```
import struct
import itertools
```

```

def write_polys(filename, polys):
    # Определяем ограничивающий параллелепипед
    flattened = list(itertools.chain(*polys))
    min_x = min(x for x, y in flattened)
    max_x = max(x for x, y in flattened)
    min_y = min(y for x, y in flattened)
    max_y = max(y for x, y in flattened)

    with open(filename, 'wb') as f:
        f.write(struct.pack('<iddddi',
                            0x1234,
                            min_x, min_y,
                            max_x, max_y,
                            len(polys)))
        for poly in polys:
            size = len(poly) * struct.calcsize('<dd')
            f.write(struct.pack('<i', size+4))
            for pt in poly:
                f.write(struct.pack('<dd', *pt))

    # Вызываем с нашими данными полигонов
    write_polys('polys.bin', polys)

```

Чтобы прочесть получившиеся данные обратно, вы можете написать похожий код с использованием функции `struct.unpack()`, которая обращает операции, проделанные во время записи. Например:

```

import struct

def read_polys(filename):
    with open(filename, 'rb') as f:
        # Читаем заголовок
        header = f.read(40)
        file_code, min_x, min_y, max_x, max_y, num_polys = \
            struct.unpack('<iddddi', header)

    polys = []
    for n in range(num_polys):
        pbytes, = struct.unpack('<i', f.read(4))
        poly = []
        for m in range(pbytes // 16):
            pt = struct.unpack('<dd', f.read(16))
            poly.append(pt)
        polys.append(poly)
    return polys

```

Хотя этот код работает, он представляет собой довольно-таки беспорядочный набор небольших операций чтения, распаковки структур и

т.п. Если такой код используется для обработки реального файла с данными, он быстро станет ещё более запутанным. Это делает очевидным необходимость поиска альтернативного решения, которое могло бы упростить некоторые шаги и позволило бы программисту сосредоточиться на более важных вещах.

В оставшейся части этого рецепта мы шаг за шагом построим достаточно продвинутое решение для интерпретации бинарных данных. Наша цель — предоставить программисту возможность передать высокоуровневую спецификацию формата файла, а все детали чтения и распаковки данных переместить в «подкапотную» часть. Заранее предупреждаем, что нижеследующий код будет самым продвинутым примером во всей книге. Он использует различные приёмы объектно-ориентированного программирования и метaprogramмирования. Рекомендуем вам внимательно прочитать раздел «Обсуждение», обращая внимание на ссылки на другие рецепты.

Во-первых, при чтении бинарных данных наиболее типичный случай — это присутствие в файле заголовков и других структур данных. Хотя модуль *struct* может распаковать эти данные в кортеж, ещё один способ представить такую информацию — это использование класса. Вот пример кода:

```
import struct

class StructField:
    ...

    Дескриптор, представляющий простое поле структуры
    ...

    def __init__(self, format, offset):
        self.format = format
        self.offset = offset
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            r = struct.unpack_from(self.format,
                                  instance._buffer, self.offset)
            return r[0] if len(r) == 1 else r

class Structure:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)
```

Этот код использует дескриптор для представления каждого поля структуры.

Каждый дескриптор содержит совместимый со *struct* формат кода вместе с байтовым сдвигом используемого буфера памяти. В методе *__get__()* функция *struct.unpack_from()* используется для распаковки значения из буфера без необходимости делать дополнительные срезы или копии.

Класс *Structure* просто служит базовым классом (суперклассом), который принимает некие байтовые данные и сохраняет их в буфере памяти, используемом дескриптором *StructField*. Функция *memoryview()* в этом классе служит целям, которые мы проясним позднее.

Этот код позволит вам определить структуру как высокоуровневый класс, который отражает информацию, найденную в таблицах, которые описывают ожидаемый формат файла. Например:

```
class PolyHeader(Structure):
    file_code = StructField('<i', 0)
    min_x = StructField('<d', 4)
    min_y = StructField('<d', 12)
    max_x = StructField('<d', 20)
    max_y = StructField('<d', 28)
    num_polys = StructField('<i', 36)
```

Вот пример использования этого класса для чтения заголовка из данных о многоугольниках, которые мы записали ранее:

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader(f.read(40))
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

Это интересно, но этот подход имеет несколько раздражающих нюансов. Даже если вы получаете удобство классоподобного интерфейса, код все равно многословен и требует от пользователя определять множество низкоуровневых деталей (например, повторяющееся использование

StructField, определение сдвигов и т.п.) В получившемся классе также отсутствуют привычные удобные моменты, такие как предоставление способа вычислить общий размер структуры.

Каждый раз, когда вы сталкиваетесь с подобным излишне многословным определением класса, вы можете подумать об использовании декоратора класса или метакласса. Одна из возможностей метакласса в том, что он может быть использован для выполнения множества низкоуровневых деталей реализации, снимая это бремя с пользователя. В качестве примера рассмотрите этот метакласс и слегка переработанный класс *Structure*:

```
class StructureMeta(type):
    """
    Метакласс, который автоматически создает дескрипторы StructField
    """

    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if format.startswith('<', '>', '!', '@')):
                byte_order = format[0]
                format = format[1:]
            format = byte_order + format
            setattr(self, fieldname, StructField(format, offset))
            offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

    class Structure(metaclass=StructureMeta):
        def __init__(self, bytedata):
            self._buffer = bytedata

        @classmethod
        def from_file(cls, f):
            return cls(f.read(cls.struct_size))
```

Используя этот новый класс *Structure*, вы можете записывать определение структуры так:

```
class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        ('d', 'min_x'),
        ('d', 'min_y'),
        ('d', 'max_x'),
        ('d', 'max_y'),
        ('i', 'num_polys')
```

]

Как вы можете видеть, это определение намного компактнее. Добавленный метод класса `from_file()` также делает более удобным чтение данных из файла, снимая необходимость знать какие-либо детали о размере или структуре данных. Например:

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

Когда вы вводите в программу метакласс, вы можете встроить в него больше «интеллекта». Например, предположим, что вы хотите обеспечить поддержку вложенных бинарных структур. Вот переделанный метакласс вместе с новым дескриптором, который это поддерживает:

```
class NestedStruct:
    ...

    Дескриптор, представляющий вложенную структуру
    ...

    def __init__(self, name, struct_type, offset):
        self.name = name
        self.struct_type = struct_type
        self.offset = offset

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            data = instance._buffer[self.offset:
                                   self.offset+self.struct_type.struct_size]
            result = self.struct_type(data)
            # Сохраняем получившуюся структуру обратно в экземпляр,
            # чтобы избежать последующего вычисления заново этого шага
            setattr(instance, self.name, result)
```

```

        return result

class StructureMeta(type):
    """
    Метакласс, который автоматически создает дескрипторы StructField
    """

    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if isinstance(format, StructureMeta):
                setattr(self, fieldname,
                        NestedStruct(fieldname, format, offset))
                offset += format.struct_size
            else:
                if format.startswith('<,>,'!','@'):
                    byte_order = format[0]
                    format = format[1:]
                format = byte_order + format
                setattr(self, fieldname, StructField(format, offset))
                offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

```

В этом примере кода дескриптор *NestedStruct* используется для наложения другого определения структуры на область памяти. Он делает это путём извлечения среза изначального буфера памяти и использования его для создания экземпляра переданного типа структуры. Поскольку буфер памяти был инициализирован как *memoryview*, это извлечение среза не приводит к созданию дополнительных копий в памяти. Вместо этого оно накладывается на изначальную память. Более того, чтобы избежать повторения создания экземпляров, дескриптор сохраняет получившуюся внутреннюю структуру объекта в экземпляр, используя тот же приём, что мы описали в [рецепте 8.10.](#)

Используя эту новую формулировку, вы можете начать писать код так:

```

class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),

```

```
(Point, 'min'),
(Point, 'max'),
('i', 'num_polys')
]
```

Удивительно, но код всё еще работает так, как вы ожидаете. Например:

```
>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min
# Вложенная структура
<__main__.Point object at 0x1006a48d0>
>>> phead.min.x
0.5
>>> phead.min.y
0.5
>>> phead.max.x
7.0
>>> phead.max.y
9.2
>>> phead.num_polys
3
```

На этом этапе мы разработали фреймворк для работы с записями фиксированного размера, но что делать с компонентами различных размеров? Например, оставшаяся часть файла с многоугольниками содержит элементы различных размеров.

Первый путь — написать класс, который просто представляет кусок (чанк) бинарных данных вместе с вспомогательной функцией для интерпретирования содержимого различными способами. Это тесно связано с подходом, описанным в [рецепте 6.11](#):

```
class SizedRecord:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

    @classmethod
    def from_file(cls, f, size_fmt, includes_size=True):
        sz_nbytes = struct.calcsize(size_fmt)
        sz_bytes = f.read(sz_nbytes)
        sz, = struct.unpack(size_fmt, sz_bytes)
        buf = f.read(sz - includes_size * sz_nbytes)
        return cls(buf)
```

```

def iter_as(self, code):
    if isinstance(code, str):
        s = struct.Struct(code)
        for off in range(0, len(self._buffer), s.size):
            yield s.unpack_from(self._buffer, off)
    elif isinstance(code, StructureMeta):
        size = code.struct_size
        for off in range(0, len(self._buffer), size):
            data = self._buffer[off:off+size]
            yield code(data)

```

Метод класса `SizedRecord.from_file()` используется для чтения из файла куска (чанка) данных с префиксом, определяющим размер, что является обычным для многих форматов файлов. На вход он принимает код форматирования структуры, который содержит кодировку размера, который должен быть представлен в байтах. Необязательный аргумент `includes_size` определяет, включает ли число байтов заголовок размера или нет. Вот пример того, как вы можете использовать этот код для прочтения отдельного многоугольника из файла с многоугольниками:

```

>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.num_polys
3
>>> polydata = [ SizedRecord.from_file(f, '<i')
...     for n in range(phead.num_polys) ]
>>> polydata
[<__main__.SizedRecord object at 0x1006a4d50>,
 <__main__.SizedRecord object at 0x1006a4f50>,
 <__main__.SizedRecord object at 0x10070da90>]
>>>

```

Как показано выше, содержимое экземпляров `SizeRecord` пока еще не интерпретировано. Чтобы сделать это, используйте метод `iter_as()`, который принимает на вход код структуры формата или класс `Structure`. Это предоставляет вам немалую гибкость в том, как можно интерпретировать данные. Например:

```

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as('<dd''):
...         print(p)
...
Polygon 0
(1.0, 2.5)
(3.5, 4.0)

```

```

(2.5, 1.5)
Polygon 1
(7.0, 1.2)
(5.1, 3.0)
(0.5, 7.5)
(0.8, 9.0)
Polygon 2
(3.4, 6.3)
(1.2, 0.5)
(4.6, 9.2)
>>>

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as(Point):
...         print(p.x, p.y)
...
Polygon 0
1.0 2.5
3.5 4.0
2.5 1.5
Polygon 1
7.0 1.2
5.1 3.0
0.5 7.5
0.8 9.0
Polygon 2
3.4 6.3
1.2 0.5
4.6 9.2
>>>

```

Собирая всё вместе, представим альтернативную реализацию функции *read_polys()*:

```

class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),
        (Point, 'max'),
        ('i', 'num_polys')
    ]

def read_polys(filename):

```

```
polys = []
with open(filename, 'rb') as f:
    phead = PolyHeader.from_file(f)
    for n in range(phead.num_polys):
        rec = SizedRecord.from_file(f, '<i')
        poly = [ (p.x, p.y)
                  for p in rec.iter_as(Point) ]
        polys.append(poly)
return polys
```

Обсуждение

Этот рецепт предоставляет практическое применение различных продвинутых приёмов программирования, в том числе дескрипторов, ленивых вычислений, метаклассов, переменных класса и представлений памяти (*memoryviews*). И все они служат очень чётко определенной цели.

Главная фишка этой реализации в том, что она полностью построена на идее ленивой распаковки. Когда экземпляр *Structure* создан, *__init__()* просто создает *memoryview* предоставленных байтовых данных и больше ничего не делает. В этот момент не производится распаковка или другие связанные со структурой операции. Причина использовать этот подход в том, что вы можете быть заинтересованы только в получении нескольких конкретных частей бинарной записи. Вместо распаковки файла целиком, будут распакованы только участки, к которым осуществляется доступ.

Чтобы реализовать ленивую распаковку и упаковку значений, используется класс-дескриптор *StructField*. Каждый атрибут, который пользователь запишет в *__fields__*, конвертируется в дескриптор *StructField*, который сохраняет связанный код формата структуры и байтовый сдвиг в хранимый буфер. Метакласс *StructureMeta* — то, что автоматически создаёт эти дескрипторы при определении различных структурных классов. Главная причина использовать метакласс в том, что он очень сильно облегчает пользователю определение формата структуры, давая возможность высокоуровневого описания без необходимости волноваться о низкоуровневых деталях.

Тонкий момент использования метакласса *StructureMeta* в том, что он делает порядок байтов липким. Так что если любой атрибут определил порядок байтов (< для little endian или > для big endian), этот порядок будет применён ко всем последующим полям. Это помогает избежать излишнего ввода с клавиатуры, но также оставляет возможность переключиться на другой

порядок в середине определения. Например, если у вас что-то сложное:

```
class ShapeFile(Structure):
    _fields_ = [ ('>i', 'file_code'),
                ('20s', 'unused'),
                ('i', 'file_length'),
                ('<i', 'version'),
                ('i', 'shape_type'),
                ('d', 'min_x'),
                ('d', 'min_y'),
                ('d', 'max_x'),
                ('d', 'max_y'),
                ('d', 'min_z'),
                ('d', 'max_z'),
                ('d', 'min_m'),
                ('d', 'max_m') ]
```

Как было отмечено, использование *memoryview()* в решении позволяет избавиться от копий в памяти. Когда структуры начинают вкладываться одна в другую, представления памяти (*memoryviews*) могут быть использованы для наложения разных частей определения структуры на одну и ту же область памяти. Этот аспект решения довольно тонкий, и он касается различий работы со срезами при использовании представлений памяти и при использовании обычных байтовых массивов. Если вы извлекаете срез из байтовой строки или массива, вы обычно получаете копию данных. А с представлением памяти это не так: срезы просто накладываются на существующую память. Поэтому этот подход эффективнее.

Немалое число связанных по смыслу рецептов расширяют освещение тем, поднятых в этом решении. Обратитесь к **рецепту 8.13.** для детального рассмотрения использования дескрипторов для построения системы типов.

Рецепт 8.10. содержит информацию о лениво вычисляемых свойствах и связан с реализацией дескриптора *NestedStruct*. **Рецепт 9.19.** рассматривает пример использования метакласса для инициализации атрибутов класса похожим способом, что приведен здесь для класса *StructureMeta*. Исходный код библиотеки Python *cotypes* также может представлять интерес, поскольку реализует примерно такую же поддержку определения структур данных, вложения структур данных и похожую функциональность.

6.13. Суммирование данных и обсчёт статистики

Задача

Вам нужно обработать большие наборы данных и сгенерировать суммы или какую-то другую статистику.

Решение

Для любого анализа данных с использованием статистики, временных рядов и прочих подобных приёмов вам стоит обратиться к библиотеке [Pandas](#).

Вот пример использования *Pandas* для анализа [городской базы крыс и грызунов Чикаго](#). К моменту написания данной книги этот CSV-файл содержал около 74 000 записей:

```
>>> import pandas

>>> # Прочесть CSV-файл, пропустив последнюю строку
>>> rats = pandas.read_csv('rats.csv', skip_footer=1)
>>> rats
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74055 entries, 0 to 74054
Data columns:
Creation Date                74055 non-null values
Status                      74055 non-null values
Completion Date              72154 non-null values
Service Request Number       74055 non-null values
Type of Service Request     74055 non-null values
Number of Premises Baited   65804 non-null values
Number of Premises with Garbage 65600 non-null values
Number of Premises with Rats 65752 non-null values
Current Activity              66041 non-null values
Most Recent Action            66023 non-null values
Street Address                74055 non-null values
ZIP Code                      73584 non-null values
X Coordinate                  74043 non-null values
Y Coordinate                  74043 non-null values
Ward                          74044 non-null values
Police District                74044 non-null values
Community Area                 74044 non-null values
Latitude                      74043 non-null values
Longitude                     74043 non-null values
Location                      74043 non-null values
dtypes: float64(11), object(9)

>>> # Исследовать диапазон значений для определённого поля
>>> rats['Current Activity'].unique()
array([nan, Dispatch Crew, Request Sanitation Inspector], dtype=object)
```

```
>>> # Отфильтровать данные
>>> crew_dispatched = rats[rats['Current Activity'] == 'Dispatch Crew']
>>> len(crew_dispatched)
65676
>>>

>> # Найти 10 самых сильно заражённых крысами ZIP-кодов (районов) в Чикаго
>>> crew_dispatched['ZIP Code'].value_counts()[:10]
60647    3837
60618    3530
60614    3284
60629    3251
60636    2801
60657    2465
60641    2238
60609    2206
60651    2152
60632    2071
>>>

>>> # Группируем по дате завершения
>>> dates = crew_dispatched.groupby('Completion Date')
<pandas.core.groupby.DataFrameGroupBy object at 0x10d0a2a10>
>>> len(dates)
472
>>>

>>> # Определяем ежедневное количество
>>> date_counts = dates.size()
>>> date_counts[0:10]
Completion Date
01/03/2011      4
01/03/2012    125
01/04/2011     54
01/04/2012     38
01/05/2011     78
01/05/2012    100
01/06/2011    100
01/06/2012     58
01/07/2011      1
01/09/2012     12
>>>

>>> # Сортируем количества
>>> date_counts.sort()
>>> date_counts[-10:]
Completion Date
10/12/2012     313
10/21/2011     314
09/20/2011     316
```

```
10/26/2011      319
02/22/2011      325
10/26/2012      333
03/17/2011      336
10/13/2011      378
10/14/2011      391
10/07/2011      457
>>>
```

Да, 7 октября 2011 года у крыс был трудный денёк.

Обсуждение

Pandas — это большая библиотека, в которой намного больше возможностей, чем мы можем здесь описать. Если вам требуется анализировать большие наборы данных, группировать данные, обсчитывать статистику и т.п., то обязательно присмотритесь к ней.

В книге [Python for Data Analysis](#) также содержится много информации по этой теме.

7. Функции

Определение функций с помощью инструкции *def* — краеугольный камень всех программ. Цель этой главы состоит в том, чтобы представить несколько продвинутых и необычных определений функций и паттернов их использования. Темы рассматривают аргументы по умолчанию, функции, способные принимать любое количество аргументов или только именованные аргументы, аннотации и замыкания. Также мы рассмотрим несколько непростых задач контроля потока управления и передачи данных, которые относятся к функциям обратного вызова (коллбэкам).

7.1. Определение функций, принимающих любое количество аргументов

Задача

Вы хотите определить функцию, которая принимает любое количество аргументов.

Решение

Чтобы определить функцию, которая принимает любое количество позиционных аргументов, используйте аргумент со звёздочкой (*argument):

```
def avg(first, *rest):
    return (first + sum(rest)) / (1 + len(rest))

# Пример использования
avg(1, 2)           # 1.5
avg(1, 2, 3, 4)     # 2.5
```

Чтобы принять любое количество именованных аргументов, используйте аргумент, который начинается с **. Например:

```
import html

def make_element(name, value, **attrs):
    keyvals = [' %s="%s"' % item for item in attrs.items()]
    attr_str = ''.join(keyvals)
    element = '<{name}{attrs}>{value}</{name}>'.format(
        name=name,
        attrs=attr_str,
        value=html.escape(value))
    return element

# Пример
# Создаёт '<item size="large" quantity="6">Albatross</item>'
make_element('item', 'Albatross', size='large', quantity=6)

# Создаёт '<p>&lt;spam&gt;</p>'
make_element('p', '<spam>')
```

Здесь *attrs* — это словарь, который хранит переданные именованные аргументы (если они были предоставлены).

Если вам нужна функция, которая может принимать и любое количество позиционных, и любое количество именованных аргументов, используйте * и ** вместе. Например:

```
def anyargs(*args, **kwargs):
    print(args)      # Кортеж
    print(kwargs)    # Словарь
```

В этой функции позиционные аргументы попадают в кортеж *args*, а все

именованные аргументы — в словарь `kwargs`.

Обсуждение

Аргумент `*` может быть только последним в списке позиционных аргументов в определении функции. Аргумент `**` может быть только последним. Тонкость тут в том, что аргумент без звёздочки может идти и после аргумента со звёздочкой:

```
def a(x, *args, y):
    pass

def b(x, *args, y, **kwargs):
    pass
```

Такие аргументы известны как «обязательные именованные аргументы», они обсуждаются далее в [рецепте 7.2](#).

7.2. Определение функций, принимающих только именованные аргументы

Задача

Вам нужна функция, которая принимает только именованные аргументы.

Решение

Эту возможность легко реализовать, если вы поместите именованные аргументы после аргумента со звёздочкой или единственной звёздочки. Например:

```
def recv(maxsize, *, block):
    'Receives a message'
    pass

recv(1024, True)        # TypeError
recv(1024, block=True) # Ok
```

Этот приём может быть также использован для определения именованных аргументов в функциях, которые принимают различное количество

позиционных аргументов. Например:

```
def minum(*values, clip=None):
    m = min(values)
    if clip is not None:
        m = clip if clip > m else m
    return m

minimum(1, 5, 2, -5, 10)          # Вернёт -5
minimum(1, 5, 2, -5, 10, clip=0)  # Вернёт 0
```

Обсуждение

Обязательные именованные аргументы часто являются хорошим способом увеличить понятность кода при определении необязательных аргументов. Например, посмотрите на такой вызов:

```
msg = recv(1024, False)
```

Пользователь, который не знаком с функцией `recv`, не имеет представления о том, что означает аргумент `False`. С другой стороны, такой вызов будет намного более ясным:

```
msg = recv(1024, block=False)
```

Использование обязательных именованных аргументов часто предпочтительнее трюков с использованием `**kwargs`, поскольку они правильно показываются, когда пользователь просит помощи:

```
>>> help(recv)
Help on function recv in module __main__:
recv(maxsize, *, block)
Receives a message
```

Обязательные именованные аргументы также полезны в более продвинутых применениях. Например, они могут быть использованы для внедрения аргументов в функции, которые применяют правила использования `*args` и `**kwargs` для получения всех входных параметров. См. рецепт 9.11.

7.3. Прикрепление информационных метаданных к аргументам функций

Задача

Вы определили функцию, но хотели бы прикрепить дополнительную информацию к аргументам, чтобы другим людям было легче понять, что делает эта функция.

Решение

Аннотации аргументов могут быть полезны, чтобы помочь программистам разобраться в том, как нужно применять функцию. Например, рассмотрим такую аннотированную функцию:

```
def add(x:int, y:int) -> int:  
    return x + y
```

Интерпретатор Python не прикрепляет никакого семантического смысла к аннотациям. Это не проверки типов, они вообще никак не влияют на поведение Python. Однако они могут помочь другим людям читать исходный код и понимать, что вы имели в виду. А вот сторонние инструменты и фреймворки могут прикреплять к аннотациям семантический смысл. Также они появляются в документации:

```
>>> help(add)  
Help on function add in module __main__:  
  
add(x: int, y: int) -> int  
>>>
```

Хотя вы можете прикрепить любой объект к функции в качестве аннотации (например, числа, строки, экземпляры и т.д.), использование классов или строк имеет наибольший смысл.

Обсуждение

Аннотации функции хранятся в атрибуте функции `__annotations__`. Например:

```
>>> add.__annotations__  
{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}
```

Хотя можно придумать немало потенциальных применений аннотаций, обычно их используют для документации. Поскольку в Python нет объявлений типов, часто бывает сложно понять, что нужно передавать в функцию, когда вы просто читаете исходный код. Аннотации дают дополнительную информацию.

См. продвинутый пример в [рецепте 9.20.](#), который показывает, как использовать аннотации для реализации множественной диспетчеризации (т.е., перегруженных функций).

7.4. Возвращение функцией нескольких значений

Задача

Вы хотите, чтобы функция возвращала несколько значений.

Решение

Чтобы вернуть несколько значений из функции, просто сделайте возвращаемым значением кортеж. Например:

```
>>> def myfun()
...     return 1, 2, 3
...
>>> a, b, c = myfun()
>>> a
1
>>> b
2
>>> c
3
```

Обсуждение

Хотя это выглядит так, будто *myfun()* возвращает несколько значений, на самом деле создаётся кортеж. Это кажется немного замысловатым, но дело в том, что кортеж задаётся не скобками, а запятыми. Например:

```
>>> a = (1, 2)      # Со скобками
```

```
>>> a  
(1, 2)  
>>> b = 1, 2      # Без скобок  
>>> b  
(1, 2)  
>>>
```

При вызове функций, которые возвращают кортеж, часто результат присваивают нескольким переменным. Это просто распаковка кортежа, описанная в [рецепте 1.1](#). Возвращаемое значение также может быть присвоено одной переменной:

```
>>> x = myfun()  
>>> x  
(1, 2, 3)  
>>>
```

7.5. Определение функций с аргументами по умолчанию

Задача

Вы хотите определить функцию или метод, где один или более аргументов являются необязательными и имеют значение по умолчанию.

Решение

Определить функцию с необязательными аргументами несложно: просто пропишите значения в определении и убедитесь, что аргументы по умолчанию идут последними. Например:

```
def spam(a, b=42):  
    print(a, b)  
spam(1)          # Ok. a=1, b=42  
spam(1, 2)       # Ok. a=1, b=2
```

Если значение по умолчанию — это изменяемый (мутабельный) контейнер, такой как список, множество или словарь, используйте `None` в качестве значения по умолчанию:

```
# Использование списка в качестве значения по умолчанию
```

```
def spam(a, b=None):
    if b is None:
        b = []
    ...

```

Если вместо предоставления значения по умолчанию вы хотите написать код, который просто проверяет, передано ли в необязательном аргументе целевое значение, используйте такую идиому:

```
_no_value = object()

def spam(a, b=_no_value):
    if b is _no_value:
        print('No b value supplied')
    ...

```

Вот как эта функция себя ведёт:

```
>>> spam(1)
No b value supplied
>>> spam(1, 2)      # b = 2
>>> spam(1, None)   # b = None
>>>
```

Понаблюдайте за разницей между отсутствием переданного значения и передачей значения `None`.

Обсуждение

Определение функций с аргументами по умолчанию — несложное дело, но не без тонкостей.

Во-первых, значения, назначенные значениями по умолчанию, связываются только один раз, во время определения функции. Попробуйте поэкспериментировать:

```
>>> x = 42
>>> def spam(a, b=x):
...     print(a, b)
...
>>> spam(1)
1 42                  # Эффекта нет
>>> x = 23
>>> spam(1)
1 42
```

```
>>>
```

Заметьте, как изменение переменной *x* (которая была использована в качестве значения по умолчанию) не оказывает влияния на последующие события.

Во-вторых, значения, назначенные значениями по умолчанию, всегда должны быть неизменяемыми объектами, такими как `None`, `True`, `False`, числа или строки. Никогда не пишите такой код:

```
def spam(a, b=[ ]):      # НЕТ!
...
...
```

Если вы это сделаете, то столкнетесь со всеми возможными неприятностями, если значение по умолчанию когда-либо покинет пределы функции и будет изменено. Такие изменения навсегда поменяют значение по умолчанию и подействуют на все будущие вызовы функции. Например:

```
>>> def spam(a, b=[ ]):
...     print(b)
...     return b
...
>>> x = spam(1)
>>> x
[]
>>> x.append(99)
>>> x.append('Yow!')
>>> x
[99, 'Yow!']
>>> spam(1)      # Возвращается изменённый список!
[99, 'Yow!']
>>>
```

Вероятно, вы хотели не этого. Чтобы избежать таких проблем, лучше назначить в качестве значения по умолчанию `None` и проверить его затем в функции, как показано в решении.

Использование оператора `is` при проверке `None` — важнейшая часть этого рецепта. Некоторые делают такую ошибку:

```
def spam(a, b=None):
    if not b:          # НЕТ! Вместо этого используйте 'b is None'
        b = []
...
...
```

Хотя `None` выдает значение `False`, многие другие объекты (например, строки нулевой длины, пустые списки, кортежи и словари) ведут себя так же. Так что показанная выше проверка будет ошибочно считать некоторые входные значения отсутствующими. Например:

```
>>> spam(1)          # OK
>>> x = []
>>> spam(1, x)      # Невидимая ошибка. Значение x перезаписывается по умолчанию
>>> spam(1, 0)      # Невидимая ошибка. 0 игнорируется
>>> spam(1, '')     # Невидимая ошибка. '' игнорируется
>>>
```

Последняя часть этого рецепта — это особенно тонкий момент: функция, которая выполняет проверку, передано ли значение (любое) в необязательном аргументе. Хитрость в том, что вы не можете использовать `None`, `0` или `False` в качестве значения по умолчанию при проверке присутствия предоставленного пользователем аргумента (поскольку все они являются вполне допустимыми аргументами, и пользователь может передать их в функцию). Так что вам нужно делать проверку как-то по-другому.

Чтобы решить эту проблему, вы можете создать уникальный частный экземпляр `object`, как показано в решении (переменная `_no_value`). Затем вы проверяете предоставленный аргумент в функции, сравнивая его с этим специальным значением, чтобы узнать, передан ли аргумент или нет. Идея в том, что крайне маловероятно, что пользователь передаст в качестве входного значения экземпляр `_no_value`. Поэтому это безопасное значение для проверки того, предоставлен ли экземпляр.

Использование `object()` может показаться необычным. `object` — это класс, который является обычным базовым классом (суперклассом) практически всех объектов Python. Вы можете создавать экземпляры `object`, но они не особенно интересны, поскольку не имеют каких-то полезных методов или атрибутов (в них нет словаря экземпляра, так что вы не можете присвоить им атрибуты). В общем-то, проверка идентичности — единственная вещь, для которой они полезны. Их можно использовать в качестве специальных значений, как и показано в вышеописанном решении.

7.6. Определение анонимных функций или функций в строке (инлайновых)

Задача

Вам нужно предоставить короткую функцию обратного вызова для использования в операции типа `sort()`, но вы не хотите определять отдельную односстрочную функцию с помощью инструкции `def`. Вместо этого вам бы пригодился способ определить функцию «инлайново» (в строке).

Решение

Простые функции, которые просто вычисляют результат выражения, могут быть заменены инструкцией `lambda`. Например:

```
>>> add = lambda x, y: x + y
>>> add(2,3)
5
>>> add('hello', 'world')
'helloworld'
>>>
```

Использование `lambda` абсолютно равноценно такому примеру:

```
>>> def add(x, y):
...     return x + y
...
>>> add(2,3)
5
>>>
```

Обычно `lambda` используется в контексте какой-то другой операции, такой как сортировка или сокращение (reduction, свёртка) данных:

```
>>> names = ['David Beazley', 'Brian Jones',
...             'Raymond Hettinger', 'Ned Batchelder']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian Jones']
>>>
```

Обсуждение

Хотя `lambda` позволяет определить простую функцию, ее возможности сильно ограничены. В частности, может быть определено только одно выражение, результат которого станет возвращаемым значением. Это

значит, что никакие другие возможности языка, в т.ч. множественные инструкции, условия, итерации и обработка исключений, использоваться не могут.

Вы можете замечательно писать код на Python без использования *lambda*. Однако вы наверняка натолкнётесь на них в написанной кем-то программе, в которой используется множество маленьких функций для вычисления результатов выражений, или же в программе, которая требует от пользователей предоставлять функции обратного вызова (коллбэки).

7.7. Захват переменных в анонимных функциях

Задача

Вы определили анонимную функцию, используя *lambda*, но вы также хотите захватить (запомнить) значения некоторых переменных во время определения.

Решение

Рассмотрим поведение следующей программы:

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

А теперь задайте себе вопрос: какими будут значения *a(10)* и *b(10)*? Если вы думаете, что 20 и 30, то ошибаетесь:

```
>>> a(10)
30
>>> b(10)
30
>>>
```

Проблема в том, что значение *x*, используемое в *lambda*-выражении, является свободной переменной, которая связывается во время выполнения (в

рантайм), а не во время определения. Так что значение `x` в `lambda`-выражениях будет таким, каким ему случится быть во время выполнения. Например:

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

Если же вы хотите, чтобы анонимная функция захватывала значение во время определения и сохраняло его, используйте значение по умолчанию:

```
>>> x = 10
>>> a = lambda y, x=x: x + y
>>> x = 20
>>> b = lambda y, x=x: x + y
>>> a(10)
20
>>> b(10)
30
>>>
```

Обсуждение

Проблема, которую мы рассматриваем в этом рецепте, возникает, когда случается перемудрить с `lambda`-функциями. Например, вы можете ожидать, что при создании списка `lambda`-выражений с использованием генератора списка или цикла, `lambda`-функции запомнят итерационные переменные во время определения. Например:

```
>>> funcs = [lambda x: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
4
4
4
4
4
>>>
```

Обратите внимание, что все функции считают, что *n* имеет последнее значение, полученное в ходе итераций. Теперь сравните со следующим:

```
>>> funcs = [lambda x, n=n: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
0
1
2
3
4
>>>
```

Как вы видите, теперь функции захватывают значения *n* во время определения.

7.8. Заставляем вызываемый объект с N аргументами работать так же, как вызываемый объект с меньшим количеством аргументов

Задача

У вас есть вызываемый объект, который вы хотели бы использовать в какой-то программе Python — возможно, в качестве функции обратного вызова (коллбэка) или обработчика (хэндлера), но он принимает слишком много аргументов и при вызове возбуждает исключение.

Решение

Если вам нужно уменьшить количество аргументов функции, используйте *functools.partial()*. Функция *partial()* позволяет присваивать фиксированные значения одному или более аргументам, что уменьшает количество аргументов, которые должны быть переданы в последующих вызовах.

Например, у вас есть вот такая функция:

```
def spam(a, b, c, d):
    print(a, b, c, d)
```

А теперь попробуем *partial()*, чтобы зафиксировать значения некоторых аргументов:

```
>>> from functools import partial
>>> s1 = partial(spam, 1)           # a = 1
>>> s1(2, 3, 4)
1 2 3 4
>>> s1(4, 5, 6)
1 2 5 6
>>> s2 = partial(spam, d=42)       # d = 42
>>> s2(1, 2, 3)
1 2 3 42
>>> s2(4, 5, 5)
4 5 5 42
>>> s3 = partial(spam, 1, 2, d=42) # a = 1, b = 2, d = 42
>>> s3(3)
1 2 3 42
>>> s3(4)
1 2 4 42
>>> s3(5)
1 2 5 42
```

Понаблюдайте, как *partial()* фиксирует значения некоторых аргументов и возвращает новый вызываемый объект. Этот новый вызываемый объект принимает пока ещё не получившие значения аргументы, объединяя их с аргументами, переданными в *partial()*, и передает всё в изначальную функцию.

Обсуждение

Этот рецепт на самом деле связан с решением задачи обеспечения совместной работы, казалось бы, несовместимого кода. Проиллюстрируем это серией примеров.

Первый пример. Предположим, что у вас есть список точек, представленных как кортежи координат (x, y). Вы можете использовать такую функцию для вычисления расстояния между двумя точками:

```
points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

import math
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.hypot(x2 - x1, y2 - y1)
```

А теперь предположим, что вы хотите отсортировать все точки по их расстоянию до какой-то другой точки. Метод списков `sort()` принимает аргумент `key`, который может быть использован для настройки поиска, но он работает только с функциями, которые принимают один аргумент (то есть `distance()` не подходит). Вот как вы можете использовать `partial()`, чтобы решить эту проблему:

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance, pt))
>>> points
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

Развивая эту идею, заметим, что `partial()` часто может использоваться для настройки сигнатур аргументов функций обратного вызова, используемых в других библиотеках. Например, вот пример кода, который использует `multiprocessing` для асинхронного вычисления результата, который передается функции обратного вызова, которая принимает результат и необязательный аргумент настройки логирования:

```
def output_result(result, log=None):
    if log is not None:
        log.debug('Got: %r', result)

# Функция-пример
def add(x, y):
    return x + y

if __name__ == '__main__':
    import logging
    from multiprocessing import Pool
    from functools import partial

    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('test')
    p = Pool()
    p.apply_async(add, (3, 4), callback=partial(output_result, log=log))
    p.close()
    p.join()
```

При передаче функции обратного вызова с использованием `apply_async()`, дополнительный аргумент настройки логирования передается с использованием `partial()`. А `multiprocessing` просто вызывает функцию обратного вызова (коллбэк) с единственным значением.

В качестве похожего примера рассмотрим задачу написания сетевых серверов. Модуль `socketserver` позволяет сделать это без особого труда. Например, вот простой эхо-сервер:

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        for line in self.rfile:
            self.wfile.write(b'GOT:' + line)

serv = TCPServer(('', 15000), EchoHandler)
serv.serve_forever()
```

Предположим, однако, что вы хотите наделить класс `EchoHandler` методом `__init__()`, который принимает дополнительный конфигурирующий аргумент. Например:

```
class EchoHandler(StreamRequestHandler):
    # ack – это добавленный обязательный именованный аргумент.
    # *args, **kwargs – это любые обычные предоставленные параметры
    # (которые переданы)
    def __init__(self, *args, ack, **kwargs):
        self.ack = ack
        super().__init__(*args, **kwargs)
    def handle(self):
        for line in self.rfile:
            self.wfile.write(self.ack + line)
```

Если вы внесёте это изменение, вы обнаружите, что больше нет очевидного пути вставить его в класс `TCPServer`. На самом деле вы обнаружите, что код начал возбуждать такие исключения:

```
Exception happened during processing of request from ('127.0.0.1', 59834)
Traceback (most recent call last):
...
TypeError: __init__() missing 1 required keyword-only argument: 'ack'
```

На первый взгляд кажется невозможным исправить этот код без попыток поправить исходник `socketserver` или ещё какого-то странного обходного решения. Однако задача легко решается с помощью `partial()` — используйте её, чтобы предоставить значение аргумента `ack`:

```
from functools import partial
```

```
serv = TCPServer(('', 15000), partial(EchoHandler, ack=b'RECEIVED:'))
serv.serve_forever()
```

В этом примере определение аргумента `ack` в методе `__init__()` может показаться немного странным, но он определяется как обязательный именованный аргумент. Это подробно рассматривается в рецепте 7.2.

Иногда функциональность `partial()` заменяется `lambda`-выражением. Например, в предыдущем примере можно применить такие инструкции:

```
points.sort(key=lambda p: distance(pt, p))

p.apply_async(add, (3, 4), callback=lambda result: output_result(result,

serv = TCPServer(('', 15000),
                  lambda *args, **kwargs: EchoHandler(*args,
                                                       ack=b'RECEIVED:',
                                                       **kwargs))
```

Этот код работает, но он более многословен и может запутать того, кто его читает. Использование `partial()` более явно сообщает о вашем намерении (передать значения некоторым аргументам).

7.9. Замена классов с одним методом функциями

Задача

У вас есть класс, который определяет только один метод, кроме `__init__()`. Однако для упрощения вашего кода вы бы хотели заменить его на простую функцию.

Решение

Во многих случаях классы с одним методом могут быть превращены в функции с помощью замыканий. Рассмотрите, например, следующий класс, который позволяет пользователю загружать страницы по URL с использованием некой шаблонной схемы:

```
from urllib.request import urlopen
```

```
class UrlTemplate:  
    def __init__(self, template):  
        self.template = template  
    def open(self, **kwargs):  
        return urlopen(self.template.format_map(kwargs))  
  
# Пример использования. Скачать данные об акциях с Yahoo  
yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')  
for line in yahoo.open(names='IBM,AAPL,FB', fields='sl1c1v'):  
    print(line.decode('utf-8'))
```

Этот класс может быть заменен намного более простой функцией:

```
def urltemplate(template):  
    def opener(**kwargs):  
        return urlopen(template.format_map(kwargs))  
    return opener  
  
# Пример использования  
yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')  
for line in yahoo(names='IBM,AAPL,FB', fields='sl1c1v'):  
    print(line.decode('utf-8'))
```

Обсуждение

Во многих случаях единственной причиной использовать класс с одним методом является необходимость сохранять дополнительное состояние для использования в методе. Например, единственное назначение класса *UrlTemplate* заключается в сохранении значения *template*, чтобы оно могло быть использовано в методе *open()*.

Использование вложенной функции (замыкания), как показано выше, часто будет намного более элегантным решением. Замыкание – это просто функция, но с дополнительным окружением переменных, которые используются внутри функции. Ключевое преимущество замыкания в том, что оно запоминает окружение, в котором было определено. Поэтому в примере функция *opener()* запоминает значение аргумента *template()*, и использует его в последующих вызовах.

Всякий раз, когда вы пишете код и встречаетесь с задачей прикрепления дополнительного состояния к функции, вспоминайте о замыканиях. Они часто являются более минималистичным и элегантным решением, нежели

альтернатива (превращение функции в полноценный класс).

7.10. Передача дополнительного состояния с функциями обратного вызова

Задача

Вы пишете код, который опирается на использование функций обратного вызова (например, на обработчики событий, коллбэки на завершения и т.п.), но вы хотите получить функцию обратного вызова, перенесящую дополнительное состояние для использования в функции обратного вызова.

Решение

Этот рецепт относится к способу использования функций обратного вызова, который можно обнаружить во многих библиотеках и фреймворках — особенно тех, которые связаны с асинхронной обработкой. Рассмотрим следующую функцию, которая вызывает коллбэк:

```
def apply_async(func, args, *, callback):
    # Вычислить результат
    result = func(*args)

    # Вызвать коллбэк с результатом
    callback(result)
```

В реальной жизни такой код может выполнять различные типы продвинутой обработки, включающей потоки, процессы и таймеры, но в данном случае это не главное. Мы просто сосредоточимся на вызове коллбэка. Вот пример использования приведённого выше кода:

```
>>> def print_result(result):
...     print('Got:', result)
...
>>> def add(x, y):
...     return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
```

```
Got: helloworld
>>>
```

Как вы можете видеть, функция `print_result()` принимает только один аргумент, который представляет собой результат. Никакая другая информация не передаётся. Это отсутствие информации иногда может представлять собой проблему, когда вы хотите, чтобы коллбэк взаимодействовал с другими переменными или частями окружения.

Способ передать дополнительную информацию в функцию обратного вызова — это использование связанного метода вместо простой функции. Например, этот класс хранит внутренний последовательный номер, который инкрементально увеличивается каждый раз, когда получен результат:

```
class ResultHandler:
    def __init__(self):
        self.sequence = 0
    def handler(self, result):
        self.sequence += 1
        print('[{}] Got: {}'.format(self.sequence, result))
```

Чтобы использовать этот класс, вы могли бы создать экземпляр и использовать связанный метод `handler` в качестве функции обратного вызова (коллбэка):

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Got: helloworld
>>>
```

В качестве альтернативы классу вы также можете использовать для хранения состояния замыкание:

```
def make_handler():
    sequence = 0
    def handle(result):
        nonlocal sequence
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
    return handle
```

Вот пример использования такого варианта:

```
>>> handler = make_handler()
>>> apply_async(add, (2, 3), callback=handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler)
[2] Got: helloworld
>>>
```

В качестве ещё одной вариации на эту тему вы также иногда можете использовать корутину (сопрограмму) для выполнения той же задачи:

```
def make_handler():
    sequence = 0
    while True:
        result = yield
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
```

Для корутины вы можете использовать метод `send()` в качестве коллбэка:

```
>> handler = make_handler()
>>> next(handler)           # Продвигаемся к yield
>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

И последнее: вы также можете передать состояние в коллбэк, используя дополнительный аргумент и применяя функцию `partial()`. Например:

```
>>> class SequenceNo:
...     def __init__(self):
...         self.sequence = 0
...
...     def handler(result, seq):
...         seq.sequence += 1
...         print('[{}] Got: {}'.format(seq.sequence, result))
...
>>> seq = SequenceNo()
>>> from functools import partial
>>> apply_async(add, (2, 3), callback=partial(handler, seq=seq))
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=partial(handler, seq=seq))
[2] Got: helloworld
>>>
```

Обсуждение

Программы, основанные на функциях обратного вызова (коллбэках), часто подвержены риску превратиться в огромную беспорядочную кучу. Частично эта проблема возникает, потому что функция обратного вызова часто отсоединена от кода, который делает первоначальный запрос, приводящий к выполнению коллбэка. Поэтому окружение выполнения между созданием запроса и обработкой результата теряется. Если вы хотите продолжить функцию обратного вызова в процедуре из нескольких шагов, вам нужно понять, как сохранить и восстановить ассоциированное состояние.

Существует два основных подхода, которые полезны для захвата и переноса состояния. Вы можете переносить его в экземпляре (например, прикрепленном к связанному методу), или же вы можете переносить его в замыкании (вложенной функции). Из этих двух приёмов замыкания, вероятно, немного более легковесны и естественны, поскольку просто создаются из функций. Они также автоматически захватывают все использованные переменные. Это освобождает вас от необходимости беспокоиться по поводу того, какое именно состояние нужно сохранить (это автоматически определяется вашим кодом).

При использовании замыканий вам нужно осторожно обращаться с изменяемыми (мутабельными) переменными. В вышеприведённом решении объявление *nonlocal* используется для обозначения того, что переменная *sequence* изменяется изнутри коллбэка. Без этого объявления вы бы получили ошибку.

Использование корутины (сопрограммы) в качестве обработчика коллбэка интересно тем, что это тесно связано с походом с использованием замыканий. В некотором смысле он даже чище, поскольку представляет собой одну функцию. Более того, переменные можно свободно изменять и не беспокоиться об объявлениях *nonlocal*. Потенциальный недостаток в том, что корутины не так легко понять, как другие компоненты Python. Есть также несколько тонких моментов — таких, как необходимость вызывать *next()* на кортине перед тем, как её использовать. Тем не менее, корутины можно использовать и по-другому — например, для определения внутристочного коллбэка (см. следующий рецепт).

Последний приём с использованием *partial()* полезен, если вам нужно просто передать дополнительные значения в коллбэк. Иногда вместо *partial()* мы

можете достичь того же с помощью *lambda*:

```
>>> apply_async(add, (2, 3), callback=lambda r: handler(r, seq))
[1] Got: 5
>>>
```

Дополнительные примеры мы можете найти в [рецепте 7.8.](#), где показывается использование *partial()* для изменения аргументных сигнатур.

7.11. Внутристочные функции обратного вызова

Задача

Вы пишете код, в котором используются коллбэки, но вас беспокоит быстрое размножение маленьких функций и головоломность потока управления. Вы бы хотели как-то заставить код выглядеть более похожим на нормальную последовательность процедурных шагов.

Решение

Коллбэки могут быть встроены в функцию путём использования генераторов и корутин (сопрограмм). Предположим, у вас есть функция, которая выполняет какую-то работу и вызывает коллбэк (см. [рецепт 7.10.](#)):

```
def apply_async(func, args, *, callback):
    # Вычисляем результат
    result = func(*args)

    # Вызываем коллбэк с результатом
    callback(result)
```

Теперь взгляните на поддерживающий код, который использует класс *Async* и декоратор *inlined_async*:

```
from queue import Queue
from functools import wraps

class Async:
    def __init__(self, func, args):
        self.func = func
```

```
self.args = args

def inlined_async(func):
    @wraps(func)
    def wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while True:
            result = result_queue.get()
            try:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue.put)
            except StopIteration:
                break
        return wrapper
```

Эти два фрагмента кода позволяют вам встроить в строку шаги функции обратного вызова, используя инструкции *yield*. Например:

```
def add(x, y):
    return x + y

@inlined_async
def test():
    r = yield Async(add, (2, 3))
    print(r)
    r = yield Async(add, ('hello', 'world'))
    print(r)
    for n in range(10):
        r = yield Async(add, (n, n))
        print(r)
    print('Goodbye')
```

Если вы вызовете *test()*, то получите такой вывод:

```
5
helloworld
0
2
4
6
8
10
12
14
16
18
```

Если исключить специальный декоратор и использование `yield`, то вы заметите, что функции обратного вызова нигде не появляются (только «под капотом»).

Обсуждение

Этот рецепт — испытание для ваших знаний в области функций обратного вызова, генераторов и потока управления.

Во-первых, основная фишка кода с коллбэками в том, что текущее вычисление приостанавливается и возобновляется в какой-то момент времени позже (асинхронно). Когда вычисление возобновляется, для продолжения обработки выполняется коллбэк. Функция `apply_sync()` иллюстрирует важнейшие составляющие выполнения коллбэка, хотя в реальном мире процесс может быть намного сложнее (в нем могут использоваться потоки, процессы, обработчики событий и т.п.)

Идея того, что вычисление приостановится и возобновится, естественным образом отображается на модель выполнения генератора. Если точнее, то операция `yield` заставляет генератор выдавать значение и приостанавливаться. Последующие вызовы методов генератора `__next__()` или `send()` заставят его снова запуститься.

Имея это в виду, мы можем понять, что суть этого рецепта заключена в декораторе `inline_async()`. Главная идея в том, что декоратор пошагово проводит генератор через все его инструкции `yield`. Чтобы это сделать, создается и изначально наполняется значениями `None` очередь результатов. Затем инициируется цикл, в котором результат вынимается из очереди и посыпается в генератор. Это вызывает следующий `yield`, где принимается экземпляр `Async`. Затем цикл смотрит на функцию и аргументы и вызывает асинхронное вычисление `apply_sync()`. Однако наиболее хитрая часть этого вычисления в том, что вместо использования обычного коллбэка, функция обратного вызова установлена на метод очереди `put()`.

В этот момент остается открытый вопрос о том, что произойдет. Главный цикл немедленно возвращается наверх и просто выполняет операцию `get()` на очереди. Если данные присутствуют, то это должен быть результат, помещенный туда коллбэком `put()`. Если же ничего нет, операция блокируется

и ждёт, когда придёт результат. Как это может произойти — зависит от конкретной реализации функции `apply_async()`.

Если вы сомневаетесь, что такая безумная штука может работать, вы можете попробовать ее с библиотекой *multiprocessing*, и заставить асинхронные операции выполнятся в отдельных процессах:

```
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Запускаем тестовую функцию
    test()
```

Вы обнаружите, что это работает, но чтобы разобраться в потоке управления, вам потребуется немало кофе.

Приём скрытия нетривиального потока управления за генераторами можно найти повсеместно в стандартной библиотеке и сторонних пакетах.

Например, декоратор `@contextmanager` из библиотеки `contextlib` выполняет похожий безумный фокус, который через инструкцию `yield` склеивает вход в менеджер контекста и выход из него. Популярный пакет *Twisted* тоже использует похожие внутристочные коллбэки.

7.12. Доступ к переменным, определенным внутри замыкания

Задача

Вы хотите добавить в замыкание функции, которые позволят получать доступ к внутренним переменным (в том числе и доступ на изменение).

Решение

В обычном случае внутренние переменные замыкания полностью скрыты от внешнего мира. Однако вы можете предоставить доступ путём написания функций для доступа и прикрепления их к замыканию в качестве атрибутов функции. Например:

```

def sample():
    n = 0
    # Функция-замыкание
    def func():
        print('n=', n)

    # Методы доступа к n
    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value

    # Прикрепление в качестве атрибутов функции
    func.get_n = get_n
    func.set_n = set_n
    return func

```

Вот пример использования этого кода:

```

>>> f = sample()
>>> f()
n = 0
>>> f.set_n(10)
>>> f()
n = 10
>>> f.get_n()
10
>>>

```

Обсуждение

Две главные возможности языка позволяют этому рецепту работать. Во-первых, инструкции *nonlocal* делают возможным написание функций, которые изменяют внутренние переменные. Во-вторых, атрибуты функции позволяют напрямую прикреплять методы для доступа к замыканию, и они работают практически так же, как методы экземпляра (хотя классы тут не используются).

Небольшое дополнение к этому рецепту позволит замыканиям эмулировать экземпляры класса. Всё, что вам нужно, это скопировать внутренние функции в словарь экземпляра и возвратить его. Например:

```
import sys
```

```

class ClosureInstance:
    def __init__(self, locals=None):
        if locals is None:
            locals = sys._getframe(1).f_locals

        # Обновить словарь экземпляра вызываемыми объектами
        self.__dict__.update((key,value) for key, value in locals.items()
                             if callable(value) )

    # Redirect special methods
    def __len__(self):
        return self.__dict__['__len__']()

# Пример использования
def Stack():
    items = [ ]

    def push(item):
        items.append(item)
    def pop():
        return items.pop()
    def __len__():
        return len(items)

    return ClosureInstance()

```

Вот интерактивный сеанс, который показывает, как всё это работает:

```

>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
3
>>> s.pop()
'Hello'
>>> s.pop()
20
>>> s.pop()
10
>>>

```

Интересно, что этот код работает немного быстрее аналога, использующего обычное определение класса. Например, вы можете проверить производительность по сравнению с таким классом:

```
class Stack2:
```

```
def __init__(self):
    self.items = [ ]

def push(self, item):
    self.items.append(item)

def pop(self):
    return self.items.pop()

def __len__(self):
    return len(self.items)
```

Если вы это сделаете, то получите похожие результаты:

```
>>> from timeit import timeit
>>> # Тест с использованием замыканий
>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Тест с использованием класса
>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

Как показано выше, версия на базе замыкания работает на 8% быстрее. По большей части выигрыш возникает за счёт прямого доступа к переменным экземпляра. Замыкания быстрее, потому что не используют дополнительную переменную *self*.

Рэймонд Хеттингер предложил еще более [дьявольский вариант этой идеи](#). Однако если вы склоняетесь использовать что-то такое в своей программе, помните, что это просто диковинная замена настоящему классу. Например, ключевые возможности типа наследования, свойств, дескрипторов или методов класса работать не будут. Вам также придётся поплясать с бубном, чтобы заставить специальные методы работать (например, обратите внимание на реализацию метода *__len__()* в *ClosureInstance*).

И последнее: вы рискуете затруднить жизнь людям, которые будут читать ваш код и размышлять о том, почему в нем нет нормального определения класса (конечно, они также задумаются, почему ваша версия работает быстрее). Но, в любом случае, это интересный пример того, чего можно достичь, предоставляя доступ к «внутренностям» замыкания.

Добавление методов в замыкания может иметь больше смысла в задачах, в

рамках которых вам нужно делать вещи типа сброса внутреннего состояния, сброса буферов, очистки кэша или реализации какого-то механизма обратной связи.

8. Классы и объекты

Основная задача этой главы — представить рецепты распространённых паттернов программирования, относящихся к определениям классов.

Рассматриваемые темы включают создание объектов, поддерживающих обычные возможности Python, использование специальных методов, приёмы инкапсуляции, наследование, управление памятью, а также полезные паттерны проектирования.

8.1. Изменение строкового представления экземпляров

Задача

Вы хотите изменить строки, которые выдаются при выводе или просмотре экземпляров, на что-то более понятное.

Решение

Чтобы изменить строковое представление экземпляра, определите методы `__str__()` и `__repr__()`. Например:

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

Метод `__repr__()` возвращает кодовое представление экземпляра, и обычно это текст, который нужно ввести, чтобы воссоздать объект. При проверке значений встроенная функция `repr()` возвращает этот текст (равно как и

интерактивный сеанс интерпретатора). Метод `__str__()` преобразует экземпляр в строку, что и будет выводом функций `str()` и `print()`. Например:

```
>>> p = Pair(3, 4)
>>> p
Pair(3, 4)          # вывод __repr__()
>>> print(p)
(3, 4)            # вывод __str__()
>>>
```

Реализация этого рецепта также показывает, как различные строковые представления могут быть использованы при форматировании. Конкретнее, специальный код форматирования `!r` показывает, что вывод `__repr__()` должен быть использован вместо вызываемого по умолчанию `__str__()`. Вы можете попробовать это показанным выше классом:

```
>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>
```

Обсуждение

Определение `__repr__()` и `__str__()` часто является хорошей практикой, поскольку может облегчить отладку и вывод экземпляра. Например, просто печатая или логируя экземпляр, программист получит более полезную информацию о содержимом экземпляра.

Стандартная практика для вывода `__repr__()` — выдавать такой текст, чтобы `eval(repr(x)) == x`. Если это невозможно или нежелательно, то обычной практикой будет создание полезного текстового представления, заключенного между `<` и `>`. Например:

```
>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>
```

Если `__str__()` не определён, используется вывод `__repr__()`.

Использование в решении `format()` может показаться немного странным, но

код форматирования `{0.x}` определяет атрибут `x` аргумента `0`. Так, в следующей функции `0` — это аргумент `self` экземпляра:

```
def __repr__(self):
    return 'Pair({0.x!r}, {0.y!r})'.format(self)
```

В качестве альтернативы этой реализации вы можете также использовать оператор `%` и такой код:

```
def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)
```

8.2. Настройка строкового форматирования

Задача

Вы хотите, чтобы объект поддерживал кастомизированное форматирование через функцию `format()` и строковый метод.

Решение

Чтобы кастомизировать строковое форматирование, определите в классе метод `__format__()`. Например:

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
}

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

Экземпляры класса `Date` теперь поддерживают операции форматирования:

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

Обсуждение

Метод `__format__()` предоставляет доступ к функциональности Python, касающейся форматирования строк. Важно отметить, что интерпретация кодов форматирования полностью зависит от самого класса. Поэтому коды могут быть практически любыми. Например, посмотрим на следующий пример из модуля `datetime`:

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, '%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:%d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>
```

Есть определённые стандартные соглашения для форматирования встроенных типов. См. [документацию модуля string](#), в которой приведена формальная спецификация.

8.3. Создание объектов, поддерживающих протокол менеджера контекста

Задача

Вы хотите заставить ваши объекты поддерживать протокол менеджера контекста (инструкцию *with*).

Решение

Чтобы сделать объекты совместимыми с инструкцией *with*, вам нужно реализовать методы `__enter__()` и `__exit__()`. Например, рассмотрим следующий класс, который предоставляет сетевое соединение:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock
    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

Ключевая возможность этого класса в том, что он представляет сетевое соединение, но изначально ничего не делает (т.е., не устанавливает соединение). Вместо этого соединение устанавливается и закрывается по запросу, с использованием инструкции *with*. Например:

```
from functools import partial

conn = LazyConnection('www.python.org', 80)
# Соединение закрыто
with conn as s:
    # conn.__enter__() выполняется: соединение открыто
    s.send(b'GET /index.html HTTP/1.0\r\n')
    s.send(b'Host: www.python.org\r\n')
    s.send(b'\r\n')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() выполняется: соединение закрыто
```

Обсуждение

Главный принцип создания менеджера контекста в том, что вы пишете код, который будет окружён блоком инструкций согласно правилам использования инструкции *with*. Когда инструкция *with* впервые встречается интерпретатору, вызывается метод `__enter__()`. Возвращённое методом `__enter__()` значение (если оно есть) помещается в переменную, указанную с помощью квалификатора `as`. Затем выполняются инструкции в теле инструкции *with*. В конце вызывается метод `__exit__()`, чтобы всё подчистить.

Этот поток управления будет выполнен, несмотря на любые события в теле инструкции *with* – даже если будут возбуждены исключения. На самом деле три аргумента метода `__exit__()` содержат тип исключения, значение и трассировку для возбужденных исключений (если они имели место). Метод `__exit__()` может как-то использовать информацию об исключении, либо проигнорировать её, ничего не делая и возвращая `None` в качестве результата. Если `__exit__()` возвращает `True`, исключение исчезнет, как будто бы ничего и не произошло, и программа продолжит выполнение инструкций, следующих сразу за блоком *with*.

Тонкий аспект этого рецепта в том, позволяет ли класс *LazyConnection* вложенное использование соединения с несколькими инструкциями *with*. Как было показано, одновременно разрешено только одно соединение через сокет, и будет возбуждено исключение, если повторить инструкцию *with*, когда сокет уже используется. Вы можете обойти это ограничение с помощью немного отличающейся реализации:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.connections = []

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        self.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
```

```
    self.connections.pop().close()

# Пример использования
from functools import partial

conn = LazyConnection('www.python.org', 80))
with conn as s1:
    ...
    with conn as s2:
        ...
        # s1 и s2 — независимые сокеты
```

В этой второй версии класс *LazyConnection* служит своего рода фабрикой соединений. Внутри для хранения стека используется список. Когда бы ни был вызван метод `__enter__()`, он создает новое соединение и добавляет его на стек. Метод `__exit__()` просто выталкивает последнее соединение со стека и закрывает его. Это тонкий момент, но это позволяет создавать множество соединений за раз с помощью вложенных инструкций `with`, как и показано выше.

Менеджеры контекста наиболее часто используются в программах, которым нужно управлять такими ресурсами, как файлы, сетевые соединения и блокировки. Ключевая особенность этих ресурсов в том, что должны быть явно закрыты или освобождены, чтобы корректно работать. Например, если вы получаете (приобретаете, завладеваете) блокировку, то должны убедиться, что освобождаете её, иначе вы рискуете войти в дедлок. Путём реализации `__enter__()`, `__exit__()` и инструкции `with` избежать таких проблем намного проще, поскольку «подчищающий» код в методе `__exit__()` будет гарантированно выполнен в любом случае.

Альтернативную реализацию менеджеров контекста вы сможете найти в модуле `contextmanager`. Потокобезопасная версия этого рецепта ждёт вас в [рецепте 12.6](#).

8.4. Экономия памяти при создании большого количества экземпляров

Задача

Ваша программа создает большое количество (миллионы) экземпляров и использует много памяти.

Решение

Для классов, которые в основном служат простыми структурами данных, вы часто можете значительно уменьшить потребление памяти экземплярами путём добавления атрибута `__slots__` в определение класса. Например:

```
class Date:  
    __slots__ = ['year', 'month', 'day']  
    def __init__(self, year, month, day):  
        self.year = year  
        self.month = month  
        self.day = day
```

Когда вы определяете `__slots__`, Python использует намного более компактное внутреннее представление экземпляров. Вместо снабжения каждого экземпляра словарём, они создаются на базе небольшого массива фиксированного размера, похожего на кортеж или список. Атрибуты, перечисленные в спецификаторе `__slots__`, внутри отображаются на конкретные индексы в массиве. Побочный эффект использования слотов в том, что теряется возможность добавления новых атрибутов к экземплярам — у вас будет возможность использовать только атрибуты, перечисленные в спецификаторе `__slots__`.

Обсуждение

Объем сэкономленной с помощью слотов памяти варьируется в зависимости от количества и типа хранимых атрибутов. Однако в общем случае объем использования памяти сравним с тем, который бы потребовался для сохранения этих данных в кортеже. Чтобы дать вам пищу для размышлений, упомянем, что сохранение одного экземпляра `Date` без использования слотов потребует 428 байтов на 64-битной версии Python. Если определены слоты, объем падает до 156 байтов. В программах, работающих с большим количеством дат одновременно, это может привести к значительному снижению использования памяти.

Хотя слоты могут показаться возможностью, которая будет полезна всегда, вы должны сопротивляться желанию использовать их повсюду. Очень многие компоненты Python основаны на стандартной реализации на базе словарей. Также нужно отметить, что классы, которые определяют слоты, не поддерживают некоторые возможности — такие как, например,

множественное наследование. По большей части вы должны использовать слоты только в классах, которые служат структурами данных в вашей программе (то есть если ваша программа создает миллионы экземпляров какого-то класса).

Часто слоты неправильно воспринимают как инструмент для инкапсуляции, который запрещает пользователям добавлять новые атрибуты к экземплярам. Хотя это действительно является побочным эффектом использования слотов, это не было целью их появления в языке. `__slots__` изначально задумывались именно как инструмент для оптимизации.

8.5. Инкапсуляция имён в классе

Задача

Вам нужно инкапсулировать «приватные» (частные) данные в экземпляре класса, но вас беспокоит, что в Python нет контроля доступа.

Решение

Вместо того, чтобы полагаться на возможности языка по инкапсулированию данных, от программистов на Python ожидается соблюдение определённых соглашений о наименовании, касающихся намеренного использования данных и методов. Первое соглашение состоит в том, что любое имя, которое начинается с одного нижнего подчёркивания (`_`) должно рассматриваться как внутренняя реализация. Например:

```
class A:
    def __init__(self):
        self._internal = 0      # Внутренний атрибут
        self.public = 1         # Внешний атрибут

    def public_method(self):
        ...
        A public method
        ...
        ...

    def _internal_method(self):
        ...
```

Python не запрещает доступ к внутренним именам. Однако это считается неправильным, и в результате может получиться хрупкий код. Также стоит отметить, что имена, начинающиеся с нижнего подчёркивания, также используются для модулей и функций уровня модуля. Например, если вы видите имя модуля, которое начинается с нижнего подчёркивания, (например, `_socket`), то это внутренняя реализация. Похожим образом, функции уровня модуля, такие как `sys._getframe()`, должны применяться очень осторожно.

Вы можете натолкнуться на имена внутри классов, которые начинаются с двух нижних подчёркиваний (`__`). Например:

```
class B:
    def __init__(self):
        self.__private = 0
    def __private_method(self):
        ...
    def public_method(self):
        ...
    self.__private_method()
    ...
```

Использование двойного нижнего подчёркивания вызывает искажение имени в другое. Если говорить конкретно, то приватные (частные) атрибуты в представленном выше классе переименуются в `_B__private` и `_B__private_method` соответственно. Здесь вы можете спросить, зачем нужны такие искажения? Причина — наследование: такие атрибуты не могут быть переопределены через наследование. Например:

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1 # Не перегружает B.__private
    # Не перегружает B.__private_method()
    def __private_method(self):
        ...
```

Здесь частные имена `__private` и `__private_method` переименуются в `_C__private` и `_C__private_method`, которые отличаются от искаженных имён в базовом классе B.

Обсуждение

Факт того, что существует два разных соглашения (одиночное нижнее подчеркивание против двойного) для определения «приватных» атрибутов, приводит к логичному вопросу того, какой стиль вам стоит использовать. Для большей части вашего кода вы, вероятно, должны делать ваши имена непубличными с помощью одного нижнего подчеркивания. Если же, однако, вы знаете, что ваш код будет использовать подклассы, и имеет внутренние атрибуты, которые должны быть скрыты от подклассов, то используйте двойные подчеркивания.

Также стоит отметить, что иногда вы можете определить переменную, которая вступает в конфликт с зарезервированным именем. Для таких случаев нужно использовать нижнее подчёркивание в конце:

```
lambda_ = 2.0  
# Завершающий _ помогает избежать проблем с ключевым словом lambda
```

Причина не использовать нижнее подчеркивание в начале имени в этом случае заключается в том, что это позволяет избежать сомнений по поводу причины его использования (то есть использование подчёркивания в начале может быть истолковано, как указание на приватность значения). Использование одного подчёркивания в конце решает эту проблему.

8.6. Создание управляемых атрибутов

Задача

Вы хотите добавить дополнительную обработку (например, проверку типов или валидацию) в получение или присваивание значения атрибуту экземпляра.

Решение

Простой способ кастомизировать доступ к атрибуту заключается в определении свойства (property). Например, этот код определяет свойство, которое добавляет простую проверку типов к атрибуту:

```
class Person:  
    def __init__(self, first_name):  
        self.first_name = first_name
```

```

# Функция-геттер
@property
def first_name(self):
    return self._first_name

# Функция-сеттер
@first_name.setter
def first_name(self, value):
    if not isinstance(value, str):
        raise TypeError('Expected a string')
    self._first_name = value

# Функция-делитер (необязательная)
@first_name.deleter
def first_name(self):
    raise AttributeError("Can't delete attribute")

```

В представленном коде есть три относящихся друг к другу метода, которые должны иметь одинаковое имя. Первый метод — это функция-геттер (getter), она делает *first_name* свойством. Два других метода прикрепляют необязательные функции сеттер (setter) и делитер (deleter) к свойству (*property*) *first_name*. Важно подчеркнуть, что декораторы *@first_name.setter* и *@first_name.deleter* не будут определены, если *first_name* не было превращено в свойство с помощью *@property*.

Важнейшая особенность свойства в том, что оно выглядит так же, как обычный атрибут, но при попытке доступа автоматически активируются геттер, сеттер и делитер. Например:

```

>>> a = Person('Guido')
>>> a.first_name          # Вызывает геттер
'Guido'
>>> a.first_name = 42     # Вызывает сеттер
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "prop.py", line 14, in first_name
      raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>>

```

Когда вы реализуете свойство, данные (если они имеются) нужно где-то сохранять. Поэтому в методах получения и присваивания вы видите прямую

манипуляцию атрибутом `_first_name`, в котором и находятся данные. Вы также можете спросить, почему метод `__init__()` устанавливает `self.first_name`, а не `self._first_name`. В этом примере весь смысл свойства заключается в применении проверки типа при присваивании значения атрибуту. Поэтому есть вероятность, что вы также захотите провести такую проверку при инициализации. Присваивая значение `self.first_name`, операция присваивания тоже использует метод-сеттер (в противоположность обходному пути прямого доступа к `self._first_name`).

Свойства также могут быть определены для существующих методов получения и присваивания значения. Например:

```
class Person:
    def __init__(self, first_name):
        self.set_first_name(first_name)

    # Функция-геттер
    def get_first_name(self):
        return self._first_name

    # Функция-сеттер
    def set_first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Функция-делитер (необязательная)
    def del_first_name(self):
        raise AttributeError("Can't delete attribute")

    # Создание свойства из существующих методов get/set
    name = property(get_first_name, set_first_name, del_first_name)
```

Обсуждение

Свойство — это на самом деле коллекция связанных вместе методов. Если вы изучите класс со свойством, то обнаружите сырье методы `fget`, `fset` и `fdel` в атрибутах самого свойства.

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
```

```
>>>
```

Обычно вы не будете вызывать *fget* и *fset* напрямую, но они автоматически вызываются, когда происходит доступ к свойству.

Свойства должны быть использованы только в случаях, когда вы на самом деле нуждаетесь в выполнении дополнительных операций при доступе к атрибутам. Иногда программисты, пришедшие из языков типа Java, считают, что любой доступ нужно осуществлять с помощью геттеров и сеттеров, и пишут такой код:

```
class Person:
    def __init__(self, first_name):
        self.first_name = name
    @property
    def first_name(self):
        return self._first_name
    @first_name.setter
    def first_name(self, value):
        self._first_name = value
```

Не определяйте свойства, которые не ничего не добавляют (как в примере выше). Во-первых, они делают ваш код многословным и непонятным другим. Во-вторых, они сделают вашу программу намного медленнее. И последнее: с точки зрения проектирования в этом нет никакого преимущества. Если вы в будущем решите, что нужно добавить дополнительную обработку к доступу к обычному атрибуту, то просто превратите его в свойство, что не приведет к необходимости менять существующий код. Это возможно, потому что синтаксис кода, который осуществляет доступ к атрибуту, останется неизменным.

Свойства также могут быть способом определить вычисляемые атрибуты. Это атрибуты, которые не хранятся, а вычисляются по запросу. Например:

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    @property
    def area(self):
        return math.pi * self.radius ** 2
    @property
    def perimeter(self):
        return 2 * math.pi * self.radius
```

Здесь использование свойств позволяет создать единообразный интерфейс экземпляра, в котором к *radius*, *area* и *perimeter* доступ осуществляется как к простым атрибутам, в противоположность смеси простых атрибутов и вызовов методов. Например:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area           # Заметьте отсутствие ()
50.26548245743669
>>> c.perimeter      # Заметьте отсутствие ()
25.132741228718345
>>>
```

Хотя свойства дают вам элегантный интерфейс программирования, иногда вы иногда можете захотеть использовать геттеры и сеттеры напрямую.

Например:

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

Такое часто имеет место, когда код на Python интегрируется в более крупную инфраструктуру систем или программ. Например, предположим, что класс Python будет вставлен в большую распределённую систему, основанную на удалённых вызовах процедур или распределённых объектов. В таких условиях может оказаться намного легче работать с явными методами присваивания и получения значений (вызывая их как обычные методы), вместо использования свойств для скрытого совершения таких вызовов.

И последнее: не пишите код на Python, в котором много повторяющихся определений свойств. Например:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name
```

```

@first_name.setter
def first_name(self, value):
    if not isinstance(value, str):
        raise TypeError('Expected a string')
    self._first_name = value

# Повторение кода свойства, но под другим именем (плохо!)
@property
def last_name(self):
    return self._last_name

@last_name.setter
def last_name(self, value):
    if not isinstance(value, str):
        raise TypeError('Expected a string')
    self._last_name = value

```

Повторение кода ведёт к раздатому, подверженному ошибкам и уродливому коду. Есть намного лучшие пути добиться того же, используя дескрипторы или замыкания. См. рецепт 8.9. и рецепт 9.21.

8.7. Вызов метода родительского класса

Задача

Вы хотите вызвать метод родительского класса вместо метода, который был переопределён в подклассе.

Решение

Чтобы вызвать метод из родительского класса (суперкласса), используйте функцию `super()`. Например:

```

class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam()      # Вызов spam() родителя

```

Очень распространённый случай использования `super()` — это применение её к методу `__init__()`, чтобы убедиться в правильной инициализации родителей:

```

class A:
    def __init__(self):
        self.x = 0

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 1

```

Также `super()` часто используется в коде, который переопределяет один из специальных методов Python. Например:

```

class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Передача поиска атрибута внутреннему obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Передача присвоения атрибута
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
            # Вызов изначального __setattr__
        else:
            setattr(self._obj, name, value)

```

В этом коде реализация `__setattr__()` включает проверку имени. Если имя начинается с нижнего подчёркивания (`_`), он вызывает изначальную реализацию `__setattr__()` через использование `super()`. В противном случае оно делегируется внутреннему объекту `self._obj`. Это выглядит немного странно, но `super()` работает, даже если явно не указан базовый класс.

Обсуждение

Правильное использование функции `super()` — это один из самых трудных для понимания аспектов Python. Вы наверняка встретите код, который напрямую вызывает метод родительского класса:

```

class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    pass

```

```
def __init__(self):
    Base.__init__(self)
    print('A.__init__')
```

Хотя это обычно «работает», это может привести к странным проблемам в продвинутых программах, использующих множественное наследование. Например:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

Если вы запустите эту программу, то увидите, что метод *Base.__init__()* вызывается дважды:

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
>>>
```

Дублирование вызова *Base.__init__()* может не нанести вреда, но может и всё поломать. Если же вы измените код так, чтобы он использовал *super()*, всё будет работать:

```
class Base:
    def __init__(self):
        print('Base.__init__')
```

```

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__()    # Здесь только вызов только super()
        print('C.__init__')

```

При использовании этой новой версии вы обнаружите, что каждый метод `__init__()` вызывается только один раз:

```

>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>>

```

Чтобы понять, почему это работает, мы должны сделать шаг назад и обсудить, как в Python реализовано наследование. Для каждого класса, который вы определите, Python вычисляет так называемый список порядка разрешения методов (ПРМ, MRO). Список ПРМ — это просто линейно упорядоченные базовые классы. Например:

```

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class '__main__.Base'>, <class 'object'>)
>>>

```

Чтобы реализовать наследование, Python начинает с самого левого класса и проходит по классам в списке ПРМ слева направо, пока не найдёт нужный атрибут.

Определение самого списка ПРМ происходит путём использования [СЗ-линеаризации](#). Не будем погружаться в математические тонкости, но по сути это сортировка слиянием ПРМ родительских классов с тремя условиями:

- классы-потомки проверяются перед родительскими;
- множественные родительские классы проверяются по порядку;
- если для следующего класса есть два валидных выбора, выбирается класс от первого родителя.

Если честно, вам нужно знать только то, что порядок классов в списке ПРМ имеет правильный смысл для практически любой классовой иерархии, которую вы реализуете.

Когда вы используете функцию `super()`, Python продолжает свой поиск, начиная со следующего класса в ПРМ. Пока каждый переопределённый метод использует `super()` и вызывает её один раз, поток управления найдет свой путь через весь список ПРМ, и каждый метод будет вызван единожды. Вот почему вы не должны делать двойные вызовы `Base.__init__()` во втором примере.

Аспект `super()`, который может удивить — это то, что она не обязательно обращается к прямому родителю следующего в ПРМ класса, а также то, что вы можете использовать её даже с классом, не имеющим прямого родителя. Рассмотрим, например, такой класс:

```
class A:
    def spam(self):
        print('A.spam')
        super().spam()
```

Если вы попробуете его использовать, вы обнаружите, что он не работает:

```
>>> a = A()
>>> a.spam()
A.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

Но посмотрите, что случится, если вы будете использовать этот класс с множественным наследованием:

```
>>> class B:
...     def spam(self):
...         print('B.spam')
```

```
...
>>> class C(A,B):
...     pass
...
>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

Здесь вы видите, что использование `super().spam()` в классе *A* на самом деле вызвало метод *spam()* в классе *B* — классе, абсолютно никак не связанном с *A*! Это объясняется ПРМ класса *C*:

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>)
>>>
```

Использование `super()` таким способом наиболее часто можно встретить в классах-миксинах (примесях). См. [рецепт 8.13.](#) и [рецепт 8.18.](#)

Но поскольку `super()` может вызывать метод, который вы не ожидаете, есть несколько общих правил, которым вам нужно попытаться следовать. Во-первых, убедитесь, что все методы с одинаковыми именами в иерархии наследования имеют совместимые сигнатуры вызова (то есть одинаковое количество аргументов, имена аргументов). Это позволяет удостовериться, что `super()` не сломается, если попытается вызвать метод класса, который не является прямым родителем. Во-вторых, обычно является хорошей идеей убедиться, что класс верхнего уровня предоставляет реализацию метода, так что поиск, который идёт по цепи ПРМ, завершится каким-то конкретным методом.

Использование `super()` иногда становится источником дебатов в сообществе Python. Однако, при всех прочих равных, вам стоит использовать её в современном коде. Рэймонд Хеттингер написал отличный пост [“Python’s super\(\) Considered Super!”](#), в котором вы найдёте ещё больше примеров того, почему `super()` может быть суперкрутой штукой.

8.8. Расширение свойства в подклассе

Задача

Внутри подкласса вы хотите расширить функциональность свойства, определённого в родительском классе.

Решение

Рассмотрите следующий код, в котором определяется свойство:

```
class Person:
    def __init__(self, name):
        self.name = name

    # Функция-геттер
    @property
    def name(self):
        return self._name

    # Функция-сеттер
    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._name = value

    # Функция-делитер
    @name.deleter
    def name(self):
        raise AttributeError("Can't delete attribute")
```

Вот пример класса, который наследует от *Person* и расширяет свойство *name* новой функциональностью:

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
```

```
super(SubPerson, SubPerson).name.__delete__(self)
```

Вот пример использования нового класса:

```
>>> s = SubPerson('Guido')
Setting name to Guido
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "example.py", line 16, in name
      raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

Если вы хотите расширить только один из методов свойства, используйте такой код:

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```

Или, альтернативно, только для сеттера, используйте такой код:

```
class SubPerson(Person):
    @Person.name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
```

Обсуждение

Расширение свойства в подклассе заставляет столкнуться с достаточно большим количеством тонких проблем, связанных с тем фактом, что свойство определяется как коллекция из геттера, сеттера и делитера, а не как один метод. Поэтому при расширении свойства вам нужно понять, будете ли вы переопределять все методы или только один.

В первом примере все методы свойства переопределяются вместе. В каждом

методе используется `super()` для вызова предыдущей реализации. Использование `super(SubPerson, SubPerson).name.__set__(self, value)` в функции-сеттере — это не ошибка. Чтобы делегировать предыдущую реализацию сеттера, поток управления должен пройти через метод `__set__()` ранее определённого свойства `name`. Однако единственный способ получить этот метод — это доступ к нему как к переменной класса, а не как к переменной экземпляра. Это происходит в операции `super(SubPerson, SubPerson)`.

Если хотите переопределить только один из методов, недостаточно использовать `@property`. Например, вот такой код не работает:

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name
```

Если вы попробуете использовать получившийся код, вы обнаружите, что функция-сеттер полностью исчезла:

```
>>> s = SubPerson('Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 5, in __init__
    self.name = name
AttributeError: can't set attribute
>>>
```

Вместо этого вы должны были изменить код так, как показано в решении:

```
class SubPerson(Person):
    @Person.getter
    def name(self):
        print('Getting name')
        return super().name
```

Когда вы это сделаете, все ранее определённые методы свойства будут скопированы, а функция-геттер заменена. Теперь оно работает так, как ожидается:

```
>>> s = SubPerson('Guido')
>>> s.name
Getting name
```

```
'Guido'
>>> s.name = 'Larry'
>>> s.name
Getting name
'Larry'
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "example.py", line 16, in name
      raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

В этом конкретном решении нет способа заменить жестко прописанное имя класса *Person* чем-то более общим. Если вы не знаете, в каком базовом классе определено свойство, вы должны использовать решение, в котором все методы свойства переопределются, а *super()* используется для передачи управления предыдущей реализации.

Стоит отметить, что первый приём, показанный в этом рецепте, также может быть использован для расширения дескриптора, как описано в [рецепте 8.9](#). Например:

```
# Дескриптор
class String:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        instance.__dict__[self.name] = value

# Класс с дескриптором
class Person:
    name = String('name')
    def __init__(self, name):
        self.name = name

# Расширение дескриптора свойством
class SubPerson(Person):
    @property
    def name(self):
```

```
    print('Getting name')
    return super().name

@name.setter
def name(self, value):
    print('Setting name to', value)
    super(SubPerson, SubPerson).name.__set__(self, value)

@name.deleter
def name(self):
    print('Deleting name')
    super(SubPerson, SubPerson).name.__delete__(self)
```

Наконец, стоит отметить, что к тому моменту, когда вы это прочитаете, расширение сеттеров и делитеров в подклассах может быть уже как-то упрощено. Показанное решение работает, но баг, описанный на [странице](#) [багов Python](#), может быть исправлен путём реализации более ясного подхода в будущих версиях Python.

8.9. Создание нового типа атрибута класса или экземпляра

Задача

Вы хотите создать новый тип атрибута экземпляра с некой дополнительной функциональностью — например, с проверкой типа.

Решение

Если вы хотите создать полностью новый тип атрибута экземпляра, определите его функциональность в форме класса-дескриптора. Вот пример:

```
# Дескриптор атрибута для целочисленного атрибута с проверкой типа
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
```

```

def __set__(self, instance, value):
    if not isinstance(value, int):
        raise TypeError('Expected an int')
    instance.__dict__[self.name] = value

def __delete__(self, instance):
    del instance.__dict__[self.name]

```

Дескриптор — это класс, который реализует три ключевых операции доступа к атрибутам (получение, присваивания значения и удаления) в форме специальных методов `__get__()`, `__set__()` и `__delete__()`. Эти методы работают путем получения экземпляра на вход. Затем производятся манипуляции над словарём экземпляра.

Чтобы использовать дескриптор, экземпляры дескриптора размещаются в определении класса как переменные класса. Например:

```

class Point:
    x = Integer('x')
    y = Integer('y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

```

Когда вы это делаете, все попытки доступа к атрибуту дескриптора (то есть `x` или `y`) перехватываются методами `__get__()`, `__set__()` и `__delete__()`. Например:

```

>>> p = Point(2, 3)
>>> p.x                  # Вызываем Point.x.__get__(p, Point)
2
>>> p.y = 5              # Вызываем Point.y.__set__(p, 5)
>>> p.x = 2.3            # Вызываем Point.x.__set__(p, 2.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "descrip.py", line 12, in __set__
    raise TypeError('Expected an int')
TypeError: Expected an int
>>>

```

На вход каждый метод дескриптора получает экземпляр, над которым нужно провести какие-то манипуляции. Чтобы провести запрошенную операцию, соответствующим образом меняется словарь экземпляра (атрибут `__dict__`). Атрибут дескриптора `self.name` содержит ключ словаря, который используется для хранения реальных данных в словаре экземпляра.

Обсуждение

Дескрипторы являются основой «подкапотной» магии большинства возможностей классов Python, включая `@classmethod`, `@staticmethod`, `@property` и даже `__slots__`.

Путём определения дескриптора вы можете перехватить базовые операции экземпляров (получение, присваивание значения, удаление) на очень низком уровне и полностью кастомизировать то, как они работают. Это дает вам огромные возможности, и это один из самых важных инструментов, используемых создателями продвинутых библиотек и фреймворков.

Путаница с дескрипторами иногда возникает по причине того, что они могут быть определены только на уровне класса, но не на уровне экземпляра. Поэтому вот такой код работать не будет:

```
# Не работает
class Point:
    def __init__(self, x, y):
        self.x = Integer('x')      # Нет! Должна быть переменной класса
        self.y = Integer('y')
        self.x = x
        self.y = y
```

А реализация метода `__get__()` сложнее, чем может показаться:

```
# Дескриптор атрибута для целочисленного атрибута с проверкой типа
class Integer:
    ...
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
    ...
```

Причина того, что `__get__()` выглядит довольно сложным, заключается в различии между переменными экземпляра и переменными класса. Если доступ к дескриптору осуществляется как к переменной класса, аргумент `instance` имеет значение `None`. В этом случае стандартным подходом будет просто вернуть сам экземпляр дескриптора (хотя разрешается также любой тип нестандартной обработки). Например:

```

>>> p = Point(2,3)
>>> p.x           # Вызываем Point.x.__get__(p, Point)
2
>>> Point.x       # Вызываем Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>

```

Дескрипторы часто являются одним из компонентов крупного фреймворка, использующего декораторы или метаклассы. В этом случае их использование может быть практически незаметно. В качестве примера приведём более продвинутый код, основанный на дескрипторах, использующий декоратор класса:

```

# Дескриптор для атрибута с проверкой типа
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]

# Декоратор класса, который применяет его к выбранным атрибутам
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate

# Пример использования
@typeassert(name=str, shares=int, price=float)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares

```

```
self.price = price
```

Стоит подчеркнуть, что вам, вероятно, не стоит писать дескриптор, если вы хотите просто кастомизировать доступ к одному атрибуту конкретного класса. Для этого проще использовать свойство, как описано в [рецепте 8.6](#). Дескрипторы более полезны в ситуациях, где предполагается многократно переиспользовать код (то есть если вы хотите использовать функциональность, предоставленную дескриптором, в сотнях мест в вашем коде, или предоставить её в качестве возможности библиотеки).

8.10. Использование лениво вычисляемых свойств

Задача

Вы хотите определить доступный только для чтения атрибут как свойство, которое вычисляется при доступе к нему. Однако после того, как доступ произойдёт, значение должно кэшироваться и не пересчитываться при следующих запросах.

Решение

Эффективный путь определения ленивых атрибутов — это использование класса-дескриптора:

```
class lazyproperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return value
```

Чтобы использовать этот код, вы можете применить его в классе:

```
import math
```

```
class Circle:  
    def __init__(self, radius):  
        self.radius = radius  
  
    @lazyproperty  
    def area(self):  
        print('Computing area')  
        return math.pi * self.radius ** 2  
  
    @lazyproperty  
    def perimeter(self):  
        print('Computing perimeter')  
        return 2 * math.pi * self.radius
```

Вот пример использования в интерактивном сеансе:

```
>>> c = Circle(4.0)  
>>> c.radius  
4.0  
>>> c.area  
Computing area  
50.26548245743669  
>>> c.area  
50.26548245743669  
>>> c.perimeter  
Computing perimeter  
25.132741228718345  
>>> c.perimeter  
25.132741228718345  
>>>
```

Обратите внимание, что сообщения “Computing area” и “Computing perimeter” появляются только один раз.

Обсуждение

Во многих случаях цель применения лениво вычисляемых атрибутов заключается в увеличении производительности. Например, вы можете избежать вычисления значений, если только они действительно где-то не нужны. Показанное решение делает именно это, используя одну из особенностей дескриптора, чтобы реализовать функциональность самым эффективным способом.

Как показано в других рецептах (например, в [рецепте 8.9.](#)), когда дескриптор помещается в определение класса, его методы `__get__()`, `__set__()` и `__delete__()` действуются при доступе к атрибуту. Но если дескриптор

определяет только метод `__get__()`, то у него намного более слабое связывание, нежели обычно. В частности, метод `__get__()` срабатывает, только если атрибут, к которому осуществляется доступ, отсутствует в словаре экземпляра.

Класс `lazyproperty` использует это так: он заставляет метод `__get__()` сохранять вычисленное значение в экземпляре, используя то же имя, что и само свойство. С помощью этого значение сохраняется в словаре экземпляра и отключает будущие вычисления свойства. Вы можете понаблюдать за этим в таком примере:

```
>>> c = Circle(4.0)
>>> # Получение переменных экземпляра
>>> vars(c)
{'radius': 4.0}

>>> # Вычисление площади и последующий просмотр переменных
>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Обратите внимание, что доступ больше не вызывает свойство
>>> c.area
50.26548245743669

>>> # Удалить переменную и увидеть, что свойство снова задействуется
>>> del c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
>>>
```

Возможный недостаток этого рецепта в том, что вычисленное значение становится изменяемым после создания. Например:

```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

Если это проблема, вы можете использовать немного менее эффективную реализацию:

```
def lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def lazy(self):
        if hasattr(self, name):
            return getattr(self, name)
        else:
            value = func(self)
            setattr(self, name, value)
            return value
    return lazy
```

Если вы используете эту версию, вы обнаружите, что операции присваивания недоступны. Например:

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Недостаток такого подхода в том, что все операции получения значения проводятся через функцию-геттер свойства. Это менее эффективно, чем простой поиск значения в словаре экземпляра, как сделано в первом решении.

За сведениями о свойствах и управлении атрибутами обратесь к [рецепту 8.6](#).
Дескрипторы описаны в [рецепте 8.9](#).

8.11. Упрощение инициализации структур данных

Задача

Вы создаете много классов, которые служат структурами данных, но уже устали от написания повторяющихся и шаблонных функций `__init__()`.

Решение

Часто вы можете обобщить инициализацию структур данных в единственной функции `__init__()`, определённой в общем базовом классе. Например:

```
class Structure:
    # Переменная класса, которая определяет ожидаемые поля
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))
        # Устанавливает аргументы
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

    # Пример определения класса
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    class Point(Structure):
        _fields = ['x','y']

    class Circle(Structure):
        _fields = ['radius']
        def area(self):
            return math.pi * self.radius ** 2
```

Если вы будете использовать эти классы, то обнаружите, что они легко конструируются. Например:

```
>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "structure.py", line 6, in __init__
    raise TypeError('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments
```

Если вы решите реализовать поддержку именованных аргументов, то есть

несколько способов проектирования реализации. Один из них — такое отображение именованных аргументов, чтобы они соответствовали только именам атрибутов, определённым в `_fields`. Например:

```
class Structure:
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Установка всех позиционных аргументов
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Установка оставшихся именованных аргументов
        for name in self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))

    # Проверка на оставшиеся любые другие аргументы
    if kwargs:
        raise TypeError('Invalid argument(s): {}'.format(','.join(kwargs)))

# Пример использования
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, price=91.1)
    s3 = Stock('ACME', shares=50, price=91.1)
```

Другой возможный выбор — использование именованных аргументов как средства добавления дополнительных атрибутов, не определённых в `_fields`, к структуре. Например:

```
class Structure:
    # Переменная класса, которая определяет ожидаемые поля
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Установка аргументов
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Установка дополнительных аргументов (если они есть)
```

```

extra_args = kwargs.keys() - self._fields
for name in extra_args:
    setattr(self, name, kwargs.pop(name))
if kwargs:
    raise TypeError('Duplicate values for {}'.format(', '.join(kw
# Пример использования
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

s1 = Stock('ACME', 50, 91.1)
s2 = Stock('ACME', 50, 91.1, date='8/2/2012')

```

Обсуждение

Приём определения метода `__init__()` общего назначения может оказаться чрезвычайно полезным, если вы когда-либо будете писать программу, построенную на основе большого количества маленьких структур данных. Так вы напишете намного меньше кода, чем при ручном создании таких методов `__init__()`:

```

class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2

```

Тонкий аспект реализации касается механизма, который используется для присваивания значений с помощью функции `setattr()`. Вместо этого вы можете подумывать об использовании к прямому доступу к словарю экземпляра. Например:

```
class Structure:
```

```
# Переменная класса, которая определяет ожидаемые поля
_fields= []
def __init__(self, *args):
    if len(args) != len(self._fields):
        raise TypeError('Expected {} arguments'.format(len(self._fields)))
    else:
        # Установка аргументов (дополнительная)
        self.__dict__.update(zip(self._fields, args))
```

Хотя это работает, часто небезопасно делать предположения о реализации подкласса. Если подкласс решит использовать `__slots__` или обернёт конкретный атрибут в свойство (или дескриптор), прямой доступ к словарю экземпляра поломается. Приведённое в рецепте решение обладает максимально общей областью применения и не делает предположений по поводу подклассов.

Потенциальный недостаток этого приёма заключается в воздействии на документацию и справочные возможности IDE. Если пользователь вызывает справку по конкретному классу, требуемые аргументы не будут описаны обычным способом. Например:

```
>>> help(Stock)
Help on class Stock in module __main__:

class Stock(Structure)
...
| Methods inherited from Structure:
|
|     __init__(self, *args, **kwargs)
|
...
>>>
```

Многие из этих проблем могут быть исправлены путём прикрепления или принудительного использования сигнатуры типов в функции `__init__()`. См. рецепт 9.16.

Также нужно отметить, что можно автоматически инициализировать переменные экземпляра, используя вспомогательную функцию и так называемый «фреймхак». Например:

```
def init_fromlocals(self):
    import sys
    locs = sys._getframe(1).f_locals
```

```
for k, v in locs.items():
    if k != 'self':
        setattr(self, k, v)

class Stock:
    def __init__(self, name, shares, price):
        init_fromlocals(self)
```

В этом варианте функция `init_fromlocals()` использует `sys._getframe()`, чтобы подсмотреть локальные переменные вызывающего метода. Если использовать её как первый шаг в методе `__init__()`, локальные переменные будут такими же, как и переданные аргументы, и смогут быть легко использованы для установки атрибутов с такими же именами. Хотя этот подход обходит проблему получения правильной сигнатуры вызова в IDE, он работает на 50% медленнее, чем представленное в рецепте решение, и использует больше сложной «подкапотной» магии. Если вашему коду не нужна эта дополнительная способность, в большинстве случаев более простое решение работает просто отлично.

8.12. Определение интерфейса или абстрактного базового класса

Задача

Вы хотите определить класс, который будет служить интерфейсом, или абстрактный базовый класс, из которого вы сможете производить проверку типов и убеждаться, что некоторые методы реализованы в подклассах.

Решение

Чтобы определить абстрактный базовый класс, воспользуйтесь модулем `abc`. Например:

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxbytes=-1):
        pass
    @abstractmethod
    def write(self, data):
```

```
pass
```

Главная возможность абстрактного базового класса в том, что он не может напрямую порождать экземпляры. Если вы попробуете это сделать, то получите ошибку:

```
a = IStream()      # TypeError: Can't instantiate abstract class
                    # IStream with abstract methods read, write
```

Вместо этого абстрактный базовый класс предназначен для использования в качестве базового класса для других классов, от которых ожидается реализация требуемых методов. Например:

```
class SocketStream(IStream):
    def read(self, maxbytes=-1):
        ...
    def write(self, data):
        ...
```

В основном абстрактные базовые классы используются в коде, где нужно принудительно реализовать ожидаемый программный интерфейс. Например, можно посмотреть на базовый класс *IStream* как на высокоуровневую спецификацию для интерфейса, который позволяет читать и записывать данные. Код, который явно проверяет наличие этого интерфейса, может быть написан так:

```
def serialize(obj, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected an IStream')
    ...
```

Вы можете подумать, что такого рода проверка типов работает только путём создания подкласса абстрактного базового класса, но абстрактный базовый класс позволяет другим классам регистрироваться в качестве реализовывающих требуемый интерфейс. Например, вы можете сделать так:

```
import io

# Регистрируем встроенные классы ввода-вывода
# в качестве поддерживающих для нашего интерфейса
IStream.register(io.IOBase)

# Открыть обычный файл и провести проверку типа
```

```
f = open('foo.txt')
isinstance(f, IStream)      # Вернёт True
```

Нужно отметить, что `@abstractmethod` может быть также применён к статическим методам, методам класса и свойствам. Вам нужно просто убедиться, что вы применяете его в правильной последовательности. `@abstractmethod` нужно писать прямо перед определением функции, как показано тут:

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @property
    @abstractmethod
    def name(self):
        pass

    @name.setter
    @abstractmethod
    def name(self, value):
        pass

    @classmethod
    @abstractmethod
    def method1(cls):
        pass

    @staticmethod
    @abstractmethod
    def method2():
        pass
```

Обсуждение

Предварительно определенные абстрактные базовые классы можно найти во многих местах стандартной библиотеки. Модуль `collections` определяет разнообразные абстрактные базовые классы, относящиеся к контейнерам и итераторам (последовательности, отображения, множества и т.п.), библиотека `numbers` определяет абстрактные базовые классы, связанные с числовыми объектами (целые числа, числа с плавающей точкой, дроби и т.п.), библиотека `io` — связанные с управлением вводом-выводом.

Вы можете использовать предопределённые абстрактные базовые классы для выполнения более обобщённой проверки типов. Вот несколько

примеров:

```
import collections

# Проверить, является ли x последовательностью
if isinstance(x, collections.Sequence):
    ...
    ...

# Проверить, является ли x итерируемым объектом
if isinstance(x, collections.Iterable):
    ...
    ...

# Проверить, есть ли у x размер
if isinstance(x, collections.Sized):
    ...
    ...

# Проверить, является ли x отображением
if isinstance(x, collections.Mapping):
    ...
    ...
```

Стоит отметить, что на момент написания этой книги некоторые библиотечные модули не используют эти предопределённые классы так, как вы могли бы предположить. Например:

```
from decimal import Decimal
import numbers

x = Decimal('3.4')
isinstance(x, numbers.Real)      # Вернёт False
```

Хотя значение 3.4 технически является реальным числом, результат проверки типов не подтверждает этого, чтобы помочь избежать случайного смешивания обычных чисел с плавающими точкой и десятичных дробей из модуля *decimal*. Поэтому если вы используете функциональность абстрактных базовых классов, стоит аккуратно писать тесты, которые проверяют, что поведение именно таково, какое вам требуется.

Хотя абстрактные базовые классы облегчают проверку типов, это не стоит слишком часто использовать в программах. В своей основе Python является гибким динамическим языком. Попытки понаставлять повсюду принудительные ограничения типов ведут к более сложному коду, нежели необходимо. Вы должны принять гибкость Python.

8.13. Реализации модели данных или

СИСТЕМЫ ТИПОВ

Решение

Вам нужно определить различные структуры данных, но хотите установить принудительные ограничения на значения, которые можно назначить определённым атрибутам.

Решение

В этой задаче вы сталкиваетесь с необходимостью создать проверки или ассерты (assertions), которые вызываются при установке (присваивании значения) определенным атрибутам экземпляра. Чтобы сделать это, вам нужно кастомизировать установку атрибутов отдельно для каждого атрибута. Для этого нужно использовать дескрипторы.

Следующий пример иллюстрирует использование дескрипторов для реализации системы типов и фреймворка проверки значений:

```
# Базовый класс. Использует дескриптор для установки значения
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)
    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Дескриптор для принудительного определения типов
class Typed(Descriptor):
    expected_type = type(None)

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('expected ' + str(self.expected_type))
        super().__set__(instance, value)

# Дескриптор для принудительного определения значений
class Unsigned(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)

class MaxSized(Descriptor):
```

```

def __init__(self, name=None, **opts):
    if 'size' not in opts:
        raise TypeError('missing size option')
    super().__init__(name, **opts)

def __set__(self, instance, value):
    if len(value) >= self.size:
        raise ValueError('size must be < ' + str(self.size))
    super().__set__(instance, value)

```

Эти классы нужно рассматривать как базовые строительные блоки, из которых вы создаете модель данных или систему типов. Продолжая пример, приведём код, который реализует некоторые другие типы данных:

```

class Integer(Typed):
    expected_type = int

class UnsignedInteger(Integer, Unsigned):
    pass

class Float(Typed):
    expected_type = float

class UnsignedFloat(Float, Unsigned):
    pass

class String(Typed):
    expected_type = str

class SizedString(String, MaxSized):
    pass

```

Используя эти объекты типов, можно определить такой класс:

```

class Stock:
    # Определяем ограничения
    name = SizedString('name', size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

Применив ограничения, вы обнаружите, что присвоение атрибутов теперь валидируется. Например:

```

>>> s = Stock('ACME', 50, 91.1)
>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
      super().__set__(instance, value)
    File "example.py", line 23, in __set__
      raise ValueError('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "example.py", line 16, in __set__
      raise TypeError('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
      super().__set__(instance, value)
    File "example.py", line 35, in __set__
      raise ValueError('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>

```

Есть несколько приёмов для упрощения спецификации ограничений в классах. Один из них — это использование декоратора класса:

```

# Декоратор класса для применения ограничений
def check_attributes(**kwargs):
    def decorate(cls):
        for key, value in kwargs.items():
            if isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else:
                setattr(cls, key, value(key))
        return cls
    return decorate

# Пример
@check_attributes(name=SizedString(size=8),
                  shares=UnsignedInteger,
                  price=UnsignedFloat)

class Stock:
    def __init__(self, name, shares, price):

```

```
    self.name = name
    self.shares = shares
    self.price = price
```

Ещё один подход к упрощению спецификации ограничений — использование метакласса:

```
# Метакласс, который применяет проверку
class checkedmeta(type):
    def __new__(cls, clsname, bases, methods):
        # Прикрепление имён атрибутов к дескрипторам
        for key, value in methods.items():
            if isinstance(value, Descriptor):
                value.name = key
        return type.__new__(cls, clsname, bases, methods)

# Пример
class Stock(metaclass=checkedmeta):
    name = SizedString(size=8)
    shares = UnsignedInteger()
    price = UnsignedFloat()
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

Обсуждение

Этот рецепт использует несколько продвинутых приёмов, включая дескрипторы, классы-миксины (примеси), функцию `super()`, декораторы классов и метаклассы. Здесь мы не можем раскрыть эти темы, но примеры вы можете найти в других рецептах (см. [рецепты 8.9., 8.18., 9.12. и 9.19.](#)). Однако несколько тонких моментов всё же стоит осветить.

Во-первых, в базовом классе `Descriptor` есть метод `__set__()`, но нет соответствующего `__get__()`. Если дескриптор не делает ничего, кроме как извлекает значение с таким же именем из словаря экземпляра, определять `__get__()` не нужно — на самом деле это сделает программу медленнее. Поэтому этот рецепт сосредоточен только на реализации `__set__()`.

Различные классы-дескрипторы в общем проектируются на базе классов-миксинов (примесей). Например, классы `Unsigned` и `MaxSized` предназначены для смешивания с другими классами-дескрипторами, полученными от `Typed`. Чтобы обрабатывать конкретные типы данных, для получения нужной

функциональности используется множественное наследование.

Вы также заметите, что все методы `__init__()` различных дескрипторов запрограммированы так, чтобы иметь одинаковую сигнатуру вызовов, использующую именованные аргументы `**opts`. Класс `MaxSized` ищет требуемые атрибуты в `opts`, но просто передает их базовому классу `Descriptor`, который их устанавливает. Трудность композиции таких классов (и миксин в особенности) состоит в том, что вы не всегда знаете, как классы будут связаны друг с другом, или что будет вызывать функция `super()`. Поэтому вам нужно заставить всё это работать для любой возможной комбинации классов.

Определения различных типов классов, таких как `Integer`, `Float` и `String`, иллюстрируют полезный приём использования переменных класса для кастомизации реализации. Дескриптор `Typed` просто ищет атрибут `expected_type`, который предоставляется каждым из этих подклассов.

Использование декоратора класса или метакласса часто является полезным для упрощения спецификации пользователем. Вы заметите, что в этих примерах пользователь больше не должен прописывать имя атрибута больше, нежели один раз:

```
# Обычно
class Point:
    x = Integer('x')
    y = Integer('y')

# Метакласс
class Point(metaclass=checkedmeta):
    x = Integer()
    y = Integer()
```

Код декоратора класса и метакласса просто сканирует словарь класса в поиске дескрипторов. Когда дескриптор найден, они просто заполняют имя дескриптора, основываясь на значении ключа.

Из всех этих подходов решение с декоратором класса может предоставить наилучшую гибкость и ясность. Во-первых оно не рассчитывает ни на какую продвинутую магию типа метаклассов. Во-вторых, декоратор может быть легко добавлен или удалён из определения класса. Например, внутри декоратора может быть возможность просто пропустить все добавленные проверки. Это может позволить проверкам стать чем-то, что можно

выключить или выключить в зависимости от текущих потребностей (например, во время отладки выключить, а в продакшне включить).

И последнее: подход с использованием декоратора класса может быть также применён в качестве замены классам-миксинам (примесям), множественному наследованию и сложному использованию функции `super()`. Вот альтернативная реализация этого рецепта, использующая декораторы классов:

```
# Базовый класс. Использует дескриптор для установки значения
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Декоратор для применения проверки типов
def Typed(expected_type, cls=None):
    if cls is None:
        return lambda cls: Typed(expected_type, cls)

    super_set = cls.__set__
    def __set__(self, instance, value):
        if not isinstance(value, expected_type):
            raise TypeError('expected ' + str(expected_type))
        super_set(self, instance, value)
    cls.__set__ = __set__
    return cls

# Декоратор для беззнаковых значений
def Unsigned(cls):
    super_set = cls.__set__

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super_set(self, instance, value)
    cls.__set__ = __set__
    return cls

# Декоратор, разрешающий значения с определенным размером
def MaxSized(cls):
    super_init = cls.__init__
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        self.size = opts['size']
        super_init(self, name, **opts)

    cls.__init__ = __init__
    return cls
```

```

        super_init(self, name, **opts)
cls.__init__ = __init__

super_set = cls.__set__
def __set__(self, instance, value):
    if len(value) >= self.size:
        raise ValueError('size must be < ' + str(self.size))
    super_set(self, instance, value)
cls.__set__ = __set__
return cls

# Специализированные дескрипторы
@Typed(int)
class Integer(Descriptor):
    pass

@Unsigned
class UnsignedInteger(Integer):
    pass

@Typed(float)
class Float(Descriptor):
    pass

@Unsigned
class UnsignedFloat(Float):
    pass

@Typed(str)
class String(Descriptor):
    pass

@MaxSized
class SizedString(String):
    pass

```

Классы, определённые в этом альтернативном решении, работают так же, как и раньше (ранее показанный в примерах код не изменился), за исключением того, что всё работает намного быстрее. Например, простая проверка времени исполнения присваивания типизированного атрибута обнаруживает, что подход с использованием декораторов классов работает почти на 100% быстрее, чем поход на основе миксин (примесей). Теперь-то вы рады, что дочитали весь этот рецепт до конца?

8.14. Реализация собственных контейнеров

Задача

Вы хотите реализовать собственный кастомный класс, который копирует поведение обычного встроенного типа контейнера, такого как список или словарь. Однако вы не полностью уверены, что знаете, какие методы нужно реализовать.

Решение

Библиотека *collections* определяет разнообразные абстрактные базовые классы, которые чрезвычайно полезны при реализации собственных классов контейнеров. Для примера предположим, что вы хотите создать класс с поддержкой итераций. Чтобы сделать это, унаследуйте его от *collections.Iterable*, как показано тут:

```
import collections

class A(collections.Iterable):
    pass
```

Наследование от *collections.Iterable* проверяет, что вы реализовали все требуемые специальные методы. Если вы не сделаете этого, то получите ошибку при создании экземпляра:

```
>>> a = A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract methods __it
>>>
```

Чтобы исправить эту ошибку, просто дайте классу требуемый метод *__iter__()* и реализуйте его так, как хотите (см. [рецепты 4.2. и 4.7.](#))

Другие важные классы, определённые в *collections*, это *Sequence*, *MutableSequence*, *Mapping*, *MutableMapping*, *Set* и *MutableSet*. Многие из этих классов формируют иерархии с увеличивающими уровнями функциональности (одна из таких иерархий — *Container*, *Iterable*, *Sized*, *Sequence* и *MutableSequence*). Ещё раз: просто создайте экземпляр любого из этих классов, чтобы увидеть, какие методы нужны, чтобы реализовать собственный контейнер с требуемым поведением:

```
>>> import collections
>>> collections.Sequence()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with \
abstract methods __getitem__, __len__
>>>
```

Вот простой пример класса, который реализует предшествующие методы, чтобы создать последовательность, в которой элементы хранятся в отсортированном порядке (это не самая эффективная реализация, но она иллюстрирует общую идею):

```
import collections
import bisect

class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is None else []

    # Требуемые методы последовательности
    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Метод для добавления элемента в правильное место
    def add(self, item):
        bisect.insort(self._items, item)
```

Вот пример использования этого класса:

```
>>> items = SortedItems([5, 1, 3])
>>> list(items)
[1, 3, 5]
>>> items[0]
1
>>> items[-1]
5
>>> items.add(2)
>>> list(items)
[1, 2, 3, 5]
>>> items.add(-10)
>>> list(items)
[-10, 1, 2, 3, 5]
>>> items[1:4]
[1, 2, 3]
```

```
>>> 3 in items
True
>>> len(items)
5
>>> for n in items:
...
print(n)
...
-10
1
2
3
5
>>>
```

Как вы можете видеть, экземпляры *SortedItems* ведут себя в точности как обычная последовательность и поддерживают все обычные операции, включая индексирование, итерирование, *len()*, проверку на содержание (оператор *in*) и даже извлечение срезов.

А вот модуль *bisect*, использованный в этом рецепте, даёт удобный способ поддерживать отсортированность элементов в списке. Поскольку *bisect.insort()* вставляет элемент в список, последовательность остается отсортированной.

Обсуждение

Наследование от одного из абстрактных базовых классов из *collections* позволяет удостовериться, что ваш собственный контейнер реализует все требуемые методы, которые нужны контейнеру. Также наследование упрощает проверку типов.

Например, ваш собственный контейнер пройдёт проверки типов:

```
>>> items = SortedItems()
>>> import collections
>>> isinstance(items, collections.Iterable)
True
>>> isinstance(items, collections.Sequence)
True
>>> isinstance(items, collections.Container)
True
>>> isinstance(items, collections.Sized)
True
>>> isinstance(items, collections.Mapping)
False
```

```
>>>
```

Многие абстрактные базовые классы из *collections* также предоставляют дефолтные реализации обычных методов контейнеров. Предположим, например, что у вас есть класс, который наследует от *collections.MutableSequence*:

```
class Items(collections.MutableSequence):
    def __init__(self, initial=None):
        self._items = list(initial) if initial is None else []

    # Required sequence methods
    def __getitem__(self, index):
        print('Getting:', index)
        return self._items[index]

    def __setitem__(self, index, value):
        print('Setting:', index, value)
        self._items[index] = value

    def __delitem__(self, index):
        print('Deleting:', index)
        del self._items[index]

    def insert(self, index, value):
        print('Inserting:', index, value)
        self._items.insert(index, value)

    def __len__(self):
        print('Len')
        return len(self._items)
```

Если вы создадите экземпляр *Items*, то вы обнаружите, что он поддерживает практически все основные методы (например, *append()*, *remove()*, *count()* и т.д.). Эти методы реализованы таким образом, что они используют только требуемые методы. Вот интерактивный сеанс, который демонстрирует это:

```
>>> a = Items([1, 2, 3])
>>> len(a)
Len
3
>>> a.append(4)
Len
Inserting: 3 4
>>> a.append(2)
Len
Inserting: 4 2
```

```
>>> a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
>>> a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>
```

Этот рецепт — лишь небольшой экскурс в функциональность абстрактных базовых классов Python. Модуль *numbers* предоставляет похожую коллекцию абстрактных классов, связанных с числовыми типами данных. См. рецепт [8.12.](#), чтобы получить больше сведений о создании собственных абстрактных базовых классов.

8.15. Делегирование доступа к атрибуту

Задача

Вы хотите, чтобы экземпляр делегировал содержащемуся внутри экземпляру доступ к атрибуту — возможно, в качестве альтернативы наследованию (или чтобы реализовать прокси).

Решение

Если не усложнять, делегирование — это паттерн программирования, который подразумевает передачу ответственности за реализацию конкретной операции другому объекту. В простейшей форме это часто выглядит как-то так:

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass
```

```

class B:
    def __init__(self):
        self._a = A()

    def spam(self, x):
        # Делегирование внутреннему экземпляру self._a
        return self._a.spam(x)

    def foo(self):
        # Делегирование внутреннему экземпляру self._a
        return self._a.foo()

    def bar(self):
        pass

```

Если нужно делегировать только пару методов, написать код типа вышеприведённого будет несложно. Однако если нужно делегировать много методов, существует альтернативный подход с определением метода `__getattr__()`:

```

class A:
    def spam(self, x):
        pass

    def foo(self):
        pass

class B:
    def __init__(self):
        self._a = A()

    def bar(self):
        pass

# Показывает все методы, определённые на классе A
def __getattr__(self, name):
    return getattr(self._a, name)

```

Метод `__getattr__()` — это «общая ловушка» для обращений к атрибутам. Это метод, который вызывается, когда программа пытается обратиться к несуществующему атрибуту. В приведённом выше коде она перехватит операции доступа к неопределённым методам в классе *B* и просто делегирует их *A*. Например:

```

b = B()
b.bar()      # Вызывает B.bar() (существует на B)
b.spam(42)   # Вызывает B.__getattr__('spam') и делегирует A.spam

```

Еще один пример делегирования — это реализация прокси. Например:

```
# Класс-прокси, который оборачивается вокруг другого объекта,
# но показывает его публичные атрибуты

class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Делегирует поиск атрибутов внутреннему obj
    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    # Делегирует присвоение атрибутов
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    # Делегирует удаление атрибутов
    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)
```

Чтобы использовать этот прокси-класс, просто оберните им другой экземпляр. Например:

```
class Spam:
    def __init__(self, x):
        self.x = x
    def bar(self, y):
        print('Spam.bar:', self.x, y)

# Создаём экземпляр
s = Spam(2)

# Создаём прокси вокруг него
p = Proxy(s)

# Access the proxy
print(p.x)                  # Выводит 2
p.bar(3)                     # Выводит "Spam.bar: 2 3"
```

```
p.x = 37          # Меняет s.x на 37
```

Путём кастомизации реализации методов доступа к атрибутам вы можете настроить поведение прокси (заставить его логировать доступ, разрешить доступ только на чтение и т.д.)

Обсуждение

Делегирование иногда используется в качестве альтернативы наследованию. Например, вместо написания такого кода:

```
class A:  
    def spam(self, x):  
        print('A.spam', x)  
  
    def foo(self):  
        print('A.foo')  
  
class B(A):  
    def spam(self, x):  
        print('B.spam')  
        super().spam(x)  
  
    def bar(self):  
        print('B.bar')
```

Вы можете написать решение с использованием делегирования:

```
class A:  
    def spam(self, x):  
        print('A.spam', x)  
  
    def foo(self):  
        print('A.foo')  
  
class B:  
    def __init__(self):  
        self._a = A()  
  
    def spam(self, x):  
        print('B.spam', x)  
        self._a.spam(x)  
  
    def bar(self):  
        print('B.bar')  
  
    def __getattr__(self, name):
```

```
    return getattr(self._a, name)
```

Такое использование делегирования часто полезно в ситуациях, когда прямое наследование может не иметь смысла, или когда вы хотите лучше контролировать отношения между объектами (например, показывать наружу некоторые методы, реализовывать интерфейсы и т.п.)

При использовании делегирования для реализации прокси, нужно держать в голове несколько важных деталей. Во-первых, метод `__getattr__()` – это на самом деле «запасной» метод, который вызывается, только если атрибут не найден. Поэтому когда запрашивается доступ к атрибутам самого экземпляра прокси (например, атрибут `_obj`), этот метод вызываться не будет. Во-вторых, методы `__setattr__()` и `__getattribute__()` требуют добавления дополнительной логики, чтобы отделить атрибуты самого экземпляра прокси от атрибутов внутреннего объекта `_obj`. Общепринятое условие состоит в том, что прокси делегируются только атрибутам, которые не начинаются нижнего подчеркивания (то есть прокси показывают только «публичные» атрибуты содержащегося внутри объекта).

Важно подчеркнуть, что метод `__getattribute__()` обычно не применяется к специальным методам, которые начинаются и заканчиваются двойным нижним подчеркиванием. Например, рассмотрите такой класс:

```
class ListLike:
    def __init__(self):
        self._items = []
    def __getattribute__(self, name):
        return getattr(self._items, name)
```

Если вы попытаетесь создать объект `ListLike`, то обнаружите, что он поддерживает обычные методы списков, такие как `append()` и `insert()`. Однако он не поддерживает операторы типа `len()`, поиска элемента и т.д. Например:

```
>>> a = ListLike()
>>> a.append(2)
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

Чтобы реализовать поддержку различных операторов, вы должны вручную делегировать специальные ассоциированные методы. Например:

```
class ListLike:
    def __init__(self):
        self._items = []
    def __getattr__(self, name):
        return getattr(self._items, name)

    # Добавляет специальные методы для поддержки некоторых списковых операций
    def __len__(self):
        return len(self._items)
    def __getitem__(self, index):
        return self._items[index]
    def __setitem__(self, index, value):
        self._items[index] = value
    def __delitem__(self, index):
        del self._items[index]
```

См. рецепт [11.8.](#), в котором приведён другой пример использования делегирования в контексте создания прокси-классов для удалённого вызова процедуры.

8.16. Определение более одного конструктора в классе

Задача

Вы пишете класс и хотите, чтобы пользователи могли создавать экземпляры не только лишь единственным способом, предоставленным `__init__()`.

Решение

Чтобы определить класс с более чем одним конструктором, вы должны использовать метод класса. Вот простой пример:

```
import time
```

```
class Date:  
    # Основной конструктор  
    def __init__(self, year, month, day):  
        self.year = year  
        self.month = month  
        self.day = day  
  
    # Альтернативный конструктор  
    @classmethod  
    def today(cls):  
        t = time.localtime()  
        return cls(t.tm_year, t.tm_mon, t.tm_mday)
```

Чтобы использовать альтернативный конструктор, просто вызовите его как функцию, такую как `Date.today()`. Вот пример:

```
a = Date(2012, 12, 21)    # Первичный  
b = Date.today()          # Альтернативный
```

Обсуждение

Одно из главных применений методов класса — это определение альтернативных конструкторов, как было показано в этом рецепте. Важнейшая возможность метода класса в том, что он получает класс в первом аргументе (`cls`). Вы заметите, что этот класс используется в методе для создания и возвращения конечного экземпляра. Это тонкий момент, но этот аспект методов класса позволяет им корректно работать с такими возможностями, как наследование. Например:

```
class NewDate(Date):  
    pass  
  
c = Date.today()          # Создание экземпляра Date (cls=Date)  
d = NewDate.today()       # Создание экземпляра NewDate (cls=NewDate)
```

При определении класса с множественными конструкторами, вы должны делать функцию `__init__()` максимально простой — она должна просто присваивать атрибутам значения. А вот уже альтернативные конструкторы будут вызываться при необходимости выполнения продвинутых операций.

Вместо определения отдельного метода класса, вы можете подумывать о реализации метода `__init__()` таким образом, который позволяет обрабатывать различные условия вызова. Например:

```
class Date:  
    def __init__(self, *args):  
        if len(args) == 0:  
            t = time.localtime()  
            args = (t.tm_year, t.tm_mon, t.tm_mday)  
        self.year, self.month, self.day = args
```

Хотя этот приём в некоторых случаях работает, он часто ведёт к сложному коду, который сложно понять и поддерживать. Например, эта реализация не показывает полезные строки помощи (с именами аргументов). К тому же код, который создает экземпляры *Date* будет менее ясным. Сравните:

```
a = Date(2012, 12, 21) # Ясно. Конкретная дата.  
b = Date()             # ??? Что тут происходит?  
  
# Class method version  
c = Date.today()       # Ясно. Сегодняшняя дата.
```

Как показано, *Date.today()* вызывает обычный метод *Date.__init__()* и создаёт экземпляр класса *Date()* с подходящими аргументами года, месяца и дня. При необходимости экземпляр может быть создан без вызова метода *__init__()*. Это описывается в следующем рецепте.

8.17. Создание экземпляра без вызова *init*

Задача

Вам нужно создать экземпляр, но вы по какой-то причине хотите обойти выполнение метода *__init__()*.

Решение

«Голый» экземпляр может быть создан с помощью прямого вызова метода класса *__new__()*. Например, рассмотрите такой класс:

```
class Date:  
    def __init__(self, year, month, day):  
        self.year = year  
        self.month = month  
        self.day = day
```

Вот как вы можете создать экземпляр *Date* без вызова **init()**:

```
>>> d = Date.__new__(Date)
>>> d
<__main__.Date object at 0x1006716d0>
>>> d.year
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Date' object has no attribute 'year'
>>>
```

Как вы можете видеть, получившийся экземпляр неинициализирован.

Поэтому теперь установка нужных переменных экземпляра лежит на вашей ответственности. Например:

```
>>> data = {'year':2012, 'month':8, 'day':29}
>>> for key, value in data.items():
...     setattr(d, key, value)
...
>>> d.year
2012
>>> d.month
8
>>>
```

Обсуждение

Задача обхода метода `__init__()` иногда возникает, когда экземпляры создаются нестандартным путём, таким как десериализация данных или реализация метода класса, определённого в качестве альтернативного конструктора. Например, в показанном классе *Date* некто может определить альтернативный конструктор `today()` как показано тут:

```
from time import localtime

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        d = cls.__new__(cls)
        t = localtime()
```

```
d.year = t.tm_year  
d.month = t.tm_mon  
d.day = t.tm_mday  
return d
```

При десериализации данных из JSON вы можете получить словарь, подобный этому:

```
data = { 'year': 2012, 'month': 8, 'day': 29 }
```

Если вы хотите превратить его в экземпляр *Date*, просто используйте показанный в решении приём.

При создании экземпляров нестандартным путём, обычно лучше не делать слишком много предположений по поводу их реализации. В общем случае лучше не писать код, который напрямую манипулирует внутренним словарём *__dict__*, если вы не знаете точно, что он гарантировано будет определён. В противном случае код поломается, если класс использует *__slots__*, свойства, дескрипторы или другие продвинутые приёмы. Если же вы будете использовать для присваивания значений *setattr()*, то ваш код будет настолько универсальным, насколько это возможно.

8.18. Расширение классов с помощью миксин (примесей)

Задача

У вас есть набор полезных методов, которые вы хотели бы сделать доступными в других классах, чтобы расширить их функциональность. Однако классы, в которые добавляются методы, не всегда связаны друг с другом через наследование. Поэтому вы не можете просто прикрепить методы к общему базовому классу (суперклассу).

Решение

Проблема, решаемая в этом рецепте, часто возникает в программах, где необходима кастомизация классов. Например, библиотека может предоставлять базовый набор классов вместе с набором необязательных кастомизаций, которые могут быть применены пользователями при желании.

Чтобы проиллюстрировать это, предположим, что вы заинтересованы в добавлении различных кастомизаций к объектам (например, логирования, запрета повторного присваивания, проверки типов и т.п.). Вот набор классов-миксинов (примесей), которые это делают:

```
class LoggedMappingMixin:
    """
    Добавляет логирование для операций get/set/delete в целях отладки.
    """

    __slots__ = ()

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return super().__getitem__(key)

    def __setitem__(self, key, value):
        print('Setting {} = {!r}'.format(key, value))
        return super().__setitem__(key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return super().__delitem__(key)

class SetOnceMappingMixin:
    """
    Позволяет устанавливать ключ только один раз.
    """

    __slots__ = ()
    def __setitem__(self, key, value):
        if key in self:
            raise KeyError(str(key) + ' already set')
        return super().__setitem__(key, value)

class StringKeysMappingMixin:
    """
    Запрещает ключам быть чем-то, кроме строк.
    """

    __slots__ = ()
    def __setitem__(self, key, value):
        if not isinstance(key, str):
            raise TypeError('keys must be strings')
        return super().__setitem__(key, value)
```

Сами по себе эти классы бесполезны. Если вы попытаетесь создать их экземпляры, ничего полезного не получится, разве что вы полюбуетесь на исключения. На самом деле они должны быть подмешаны к другим классам через множественное наследование. Например:

```
>>> class LoggedDict(LoggedMappingMixin, dict):
...     pass
...
>>> d = LoggedDict()
>>> d['x'] = 23
Setting x = 23
>>> d['x']
Getting x
23
>>> del d['x']
Deleting x

>>> from collections import defaultdict
>>> class SetOnceDefaultDict(SetOnceMappingMixin, defaultdict):
...     pass
...
>>> d = SetOnceDefaultDict(list)
>>> d['x'].append(2)
>>> d['y'].append(3)
>>> d['x'].append(10)
>>> d['x'] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "mixin.py", line 24, in __setitem__
      raise KeyError(str(key) + ' already set')
KeyError: 'x already set'

>>> from collections import OrderedDict
>>> class StringOrderedDict(StringKeysMappingMixin,
...                             SetOnceMappingMixin,
...                             OrderedDict):
...     pass
...
>>> d = StringOrderedDict()
>>> d['x'] = 23
>>> d[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "mixin.py", line 45, in __setitem__
      raise TypeError("keys must be strings")
TypeError: keys must be strings
>>> d['x'] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "mixin.py", line 46, in __setitem__
      __slots__ = ()
File "mixin.py", line 24, in __setitem__
  if key in self:
TypeError: 'x already set'
>>>
```

В этом примере вы заметите, что миксины объединены с другими существующими классами (*dict*, *defaultdict*, *OrderedDict*) и даже друг с другом. При объединении классы работают вместе, предоставляя желаемую функциональность.

Обсуждение

Классы-миксины (примеси) используются в различных местах стандартной библиотеки, в основном для расширения функциональности других классов. Также они являются одной из основных причин применять множественное наследование. Например, если вы пишете сетевой код, вы часто можете использовать *ThreadingMixIn* из модуля *socketserver*, чтобы добавить поддержку потоков в другие связанные с сетью классы. Например, вот многопоточный XML-RPC-сервер:

```
from xmlrpclib import SimpleXMLRPCServer
from socketserver import ThreadingMixIn
class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    pass
```

Также часто можно встретить миксины в крупных библиотеках и фреймворках — опять же, в основном для расширения функциональности существующих классов дополнительными возможностями.

У теории классов-миксин богатая история. Однако вместо погружения во все детали стоит держать в голове несколько важных моментов реализации.

Во-первых, классы-миксины (примеси) никогда не предназначались для прямого создания экземпляров. Например, ни один из приведённых в этом рецепте классов сам по себе не работает. Они должны быть подмешаны к другому классу, который реализует требуемую функциональность. Похожим образом *ThreadingMixIn* из модуля *socketserver* должен быть подмешан к подходящему классу-серверу — он не может быть использован сам по себе.

Во-вторых, классы-миксины обычно не имеют собственного состояния. Это значит, что у них нет метода *__init__()* и переменных экземпляра. В этом рецепте определение *__slots__ = ()* предназначено для строгого указания на тот факт, что классы-миксины не имеют собственных данных экземпляра.

Если вы подумываете определить класс-миксин, у которого будет метод *__init__()* и переменные экземпляра, то обратите внимание, что существует

серьезная опасность, связанная с тем фактом, что класс ничего не знает о других классах, с которыми он будет смешиваться. Поэтому все переменные экземпляра должны иметь такие имена, которые позволяют избежать конфликтов имён. Также метод `__init__()` должен быть запрограммирован правильно вызывать метод `__init__()` других классов, к которым подмешивается миксин. В общем случае это трудно реализовать, поскольку вы ничего не знаете о сигнтурах аргументов других классов. По крайней мере, вы должны реализовать нечто очень общее, используя `*args` и `**kwargs`. Если `__init__()` класса-миксина принимает какие-либо аргументы, эти аргументы должны быть определены только как именованные — и иметь такие имена, чтобы избежать конфликтов с другими аргументами. Вот возможная реализация миксина, определяющего `__init__()` и принимающего именованный аргумент:

```
class RestrictKeysMixin:
    def __init__(self, *args, _restrict_key_type, **kwargs):
        self._restrict_key_type = _restrict_key_type
        super().__init__(*args, **kwargs)

    def __setitem__(self, key, value):
        if not isinstance(key, self._restrict_key_type):
            raise TypeError('Keys must be ' + str(self._restrict_key_type))
        super().__setitem__(key, value)
```

Вот пример использования этого класса:

```
>>> class RDict(RestrictKeysMixin, dict):
...     pass
...
>>> d = RDict(_restrict_key_type=str)
>>> e = RDict([('name', 'Dave'), ('n', 37)], _restrict_key_type=str)
>>> f = RDict(name='Dave', n=37, _restrict_key_type=str)
>>> f
{'n': 37, 'name': 'Dave'}
>>> f[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 83, in __setitem__
    raise TypeError('Keys must be ' + str(self._restrict_key_type))
TypeError: Keys must be <class 'str'>
>>>
```

В этом примере вы можете заметить, что инициализация `RDict()` всё ещё принимает аргументы, которые понимает функция `dict()`. Однако есть и

дополнительный именованный аргумент `restrict_key_type`, который предоставляетяется классу-миксину.

И последнее: функция `super()` — необходимая и критически важная часть написания классов-миксин. В вышеприведённом решении классы переопределяют некоторые критически важные методы, такие как `__getitem__()` и `__setitem__()`. Однако им также нужно вызвать изначальные реализации этих методов. Использование `super()` делегируется следующему классу в порядке разрешения методов (ПРМ, MRO — method resolution order). Этот аспект рецепта, однако, неочевиден для новичков, потому что `super()` используется в классах, у которых нет родителей (на первый взгляд это может выглядеть, как ошибка). Однако в подобном определении класса:

```
class LoggedDict(LoggedMappingMixin, dict):  
    pass
```

...использование `super()` в `LoggedMappingMixin` делегируется следующему классу в списке множественного наследования. Так, вызов типа `super().__getitem__()` в `LoggedMappingMixin` на самом деле делает дополнительный шаг и вызывает `dict.__getitem__()`. Без такого поведения класс-миксин не работал бы.

Альтернативная реализация миксин подразумевает использование декораторов классов. Например, рассмотрите такой код:

```
def LoggedMapping(cls):  
    cls_getitem = cls.__getitem__  
    cls_setitem = cls.__setitem__  
    cls_delitem = cls.__delitem__  
  
    def __getitem__(self, key):  
        print('Getting ' + str(key))  
        return cls_getitem(self, key)  
  
    def __setitem__(self, key, value):  
        print('Setting {} = {!r}'.format(key, value))  
        return cls_setitem(self, key, value)  
  
    def __delitem__(self, key):  
        print('Deleting ' + str(key))  
        return cls_delitem(self, key)  
  
    cls.__getitem__ = __getitem__  
    cls.__setitem__ = __setitem__  
    cls.__delitem__ = __delitem__
```

```
    return cls
```

Эта функция применяется к определению класса как декоратор. Например:

```
@LoggedMapping
class LoggedDict(dict):
    pass
```

Если вы попробуете это в работе, то получите такое же поведение, как и ранее, но без использования множественного наследования. Вместо него декоратор просто выполняет небольшую хирургическую операцию на определении класса для замены некоторых методов. Дополнительную информацию о декораторах вы можете почерпнуть в [рецепте 9.12](#).

См. также [рецепт 8.13.](#), где приведен продвинутый способ, использующий миксины и декораторы классов одновременно.

8.19. Реализация объектов с состоянием или конечных автоматов

Задача

Вы хотите реализовать конечный автомат (объект, который может находиться в определенном количестве различных состояний), но не хотите замусоривать код большим количеством условий.

Решение

В некоторых приложениях вам могут понадобиться объекты, которые работают по-разному в зависимости от некого внутреннего состояния. Например, рассмотрим простой класс, представляющий соединение:

```
class Connection:
    def __init__(self):
        self.state = 'CLOSED'

    def read(self):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('reading')
```

```

def write(self, data):
    if self.state != 'OPEN':
        raise RuntimeError('Not open')
    print('writing')

def open(self):
    if self.state == 'OPEN':
        raise RuntimeError('Already open')
    self.state = 'OPEN'

def close(self):
    if self.state == 'CLOSED':
        raise RuntimeError('Already closed')
    self.state = 'CLOSED'

```

Эта реализация приводит к появлению нескольких трудных моментов. Во-первых, код пере усложнён большим количеством условных проверок состояния. Во-вторых, производительность страдает из-за большого количества операций (например, `read()` и `write()` всегда проверяют состояние перед выполнением).

Более элегантный подход — закодировать каждое операционное состояние как отдельный класс, а класс `Connection` заставить делегировать операции классу состояния. Например:

```

class Connection:
    def __init__(self):
        self._state(ClosedConnectionState)

    def new_state(self, newstate):
        self._state = newstate

    # Делегирует классу состояния
    def read(self):
        return self._state.read(self)

    def write(self, data):
        return self._state.write(self, data)

    def open(self):
        return self._state.open(self)

    def close(self):
        return self._state.close(self)

# Базовый класс состояния соединения
class ConnectionState:

```

```
@staticmethod
def read(conn):
    raise NotImplementedError()

@staticmethod
def write(conn, data):
    raise NotImplementedError()

@staticmethod
def open(conn):
    raise NotImplementedError()

@staticmethod
def close(conn):
    raise NotImplementedError()

# Реализация различных состояний
class ClosedConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        raise RuntimeError('Not open')

    @staticmethod
    def write(conn, data):
        raise RuntimeError('Not open')

    @staticmethod
    def open(conn):
        conn.new_state(OpenConnectionState)

    @staticmethod
    def close(conn):
        raise RuntimeError('Already closed')

class OpenConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        print('reading')

    @staticmethod
    def write(conn, data):
        print('writing')

    @staticmethod
    def open(conn):
        raise RuntimeError('Already open')

    @staticmethod
    def close(conn):
        conn.new_state(ClosedConnectionState)
```

Вот пример использования этих классов:

```
>>> c = Connection()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 10, in read
    return self._state.read(self)
  File "example.py", line 43, in read
    raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()
>>> c._state
<class '__main__.OpenConnectionState'>
>>> c.read()
reading
>>> c.write('hello')
writing
>>> c.close()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>>
```

Обсуждение

Код с большим количеством сложных проверок выполнения условий и связанных вместе состояний трудно поддерживать и понимать.

Представленное решение обходит эту проблему путём выделения индивидуальных состояний в отдельные классы.

Это может показаться немного странным, однако каждое состояние реализовано как класс со статическими методами, каждый из которых принимает экземпляр *Connection* первым аргументом. Это проектировочное решение основано на отказе от хранения любых данных экземпляра в состояниях других классов. Вместо этого все данные экземпляра должны храниться в экземпляре *Connection*. Группирование состояний в общем базовом классе часто помогает упорядочить код и убедиться, что нужные методы реализованы. Исключение *NotImplementedError* возбуждается в методах базового класса и помогает убедиться, что подклассы предоставляют реализацию требуемых методов. В качестве альтернативы вы можете рассмотреть использование абстрактного базового класса, как то описано в [рецепте 8.12](#).

Альтернативная реализация затрагивает прямое управление атриутом `__class__` в экземплярах. Рассмотрите такой пример:

```
class Connection:
    def __init__(self):
        self.new_state(ClosedConnection)

    def new_state(self, newstate):
        self.__class__ = newstate

    def read(self):
        raise NotImplementedError()

    def write(self, data):
        raise NotImplementedError()

    def open(self):
        raise NotImplementedError()

    def close(self):
        raise NotImplementedError()

class ClosedConnection(Connection):
    def read(self):
        raise RuntimeError('Not open')

    def write(self, data):
        raise RuntimeError('Not open')

    def open(self):
        self.new_state(OpenConnection)

    def close(self):
        raise RuntimeError('Already closed')

class OpenConnection(Connection):
    def read(self):
        print('reading')
    def write(self, data):
        print('writing')
    def open(self):
        raise RuntimeError('Already open')
    def close(self):
        self.new_state(ClosedConnection)
```

Основная фишка этой реализации в том, что она устраниет дополнительный слой «косвенности» (indirection). Вместо создания отдельных классов `Connection` и `ConnectionState` вы сливаете эти классы вместе. Когда меняется

состояние, экземпляр изменяет свой тип, как показано тут:

```
>>> c = Connection()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "state.py", line 15, in read
      raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()
>>> c
<__main__.OpenConnection object at 0x1006718d0>
>>> c.read()
reading
>>> c.close()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>>
```

Пуристы от объектно-ориентированного программирования могут быть оскорблены идеей простого изменения атрибута экземпляра `__class__`. Однако это технически возможно. Также это может ускорить выполнение программы, поскольку методы не используют дополнительный шаг делегирования.

И, наконец, каждый из этих приёмов полезен для реализации более сложных конечных автоматов — особенно в коде, который может послужить альтернативой огромным блокам `if-elif-else`. Например:

```
# Изначальная реализация
class State:
    def __init__(self):
        self.state = 'A'

    def action(self, x):
        if state == 'A':
            # Action for A
            ...
            state = 'B'
        elif state == 'B':
            # Action for B
            ...
            state = 'C'
        elif state == 'C':
            # Action for C
```

```

    ...
    state = 'A'

# Альтернативная реализация
class State:
    def __init__(self):
        self.new_state(State_A)

    def new_state(self, state):
        self.__class__ = state

    def action(self, x):
        raise NotImplementedError()

class State_A(State):
    def action(self, x):
        # Действие для A
        ...
        self.new_state(State_B)

class State_B(State):
    def action(self, x):
        # Действие для B
        ...
        self.new_state(State_C)

class State_C(State):
    def action(self, x):
        # Действие для C
        ...
        self.new_state(State_A)

```

Этот рецепт основан на паттерне (шаблоне) проектирования «Состояние» (State) из книги «[Приёмы объектно-ориентированного проектирования. Паттерны проектирования](#)» Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса.

8.20. Вызов метода объекта с передачей имени метода в строке

Задача

У вас есть имя метода, хранящееся в виде строки, и вы хотите вызывать этот

метод.

Решение

В простых случаях вы можете использовать `getattr()`:

```
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point({!r:}, {!r:})'.format(self.x, self.y)

    def distance(self, x, y):
        return math.hypot(self.x - x, self.y - y)

p = Point(2, 3)
d = getattr(p, 'distance')(0, 0)      # Calls p.distance(0, 0)
```

Альтернативный подход — использование `operator.methodcaller()`. Например:

```
import operator
operator.methodcaller('distance', 0, 0)(p)
```

`operator.methodcaller()` может оказаться полезным, если вы хотите искать метод по имени и раз за разом предоставлять одни и те же аргументы. Например, если вам нужно отсортировать целый список точек:

```
points = [
    Point(1, 2),
    Point(3, 0),
    Point(10, -3),
    Point(-5, -7),
    Point(-1, 8),
    Point(3, 2)
]

# Сортируем по расстоянию от (0, 0)
points.sort(key=operator.methodcaller('distance', 0, 0))
```

Обсуждение

Вызов метода на самом деле состоит из двух отдельных шагов: поиска атрибута и вызова функции. Чтобы вызвать метод, вы просто ищете атрибут, используя `getattr()`, как и для любого другого атрибута. Чтобы вызывать результат как метод, просто обращайтесь с результатом поиска как с функцией.

`operator.methodcaller()` создаёт вызываемый объект, а также фиксирует аргументы, которые будут предоставлены методу. Вам остаётся только предоставить подходящий аргумент `self`. Например:

```
>>> p = Point(3, 4)
>>> d = operator.methodcaller('distance', 0, 0)
>>> d(p)
5.0
>>>
```

Вызов методов с использованием имён, содержащихся в строках, является обычным в коде, который эмулирует объявления условий или вариантах паттерна (шаблона проектирования) «Посетитель». См. следующий рецепт, где приведён более продвинутый пример.

8.21. Реализация шаблона проектирования «Посетитель»

Задача

Вам нужно написать код, который обрабатывает сложную структуру данных, состоящую из множества различных типов объектов, каждый из которых нужно обрабатывать отдельным способом. Примером этого может послужить прохождение по древовидной структуре и выполнение различных действий в зависимости от того, какие узлы дерева встречаются по пути.

Решение

Задача, которая решается этим рецептом, часто возникает в программах, которые строят структуры данных, состоящие из большого количества разнородных объектов. Чтобы проиллюстрировать это, предположим, что вы пытаетесь написать программу, которая представляет математические выражения. Чтобы сделать это, программа может использовать классы:

```

class Node:
    pass

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

```

Эти классы могут в дальнейшем использоваться для построения вложенных структур данных:

```

# Представление 1 + 2 * (3 - 4) / 5
t1 = Sub(Number(3), Number(4))
t2 = Mul(Number(2), t1)
t3 = Div(t2, Number(5))
t4 = Add(Number(1), t3)

```

Проблема не в создании таких структур, а в написании кода, который будет их обрабатывать. Например, в таком выражении программа может захотеть выполнить любое количество операций (т.е., создать вывод, сгенерировать инструкции, выполнить перевод и т.п.)

При необходимости подключить обработку общего назначения, обычное решение заключается в реализации паттерна (шаблона проектирования)

«Посетитель» с использованием вот такого класса:

```
class NodeVisitor:  
    def visit(self, node):  
        methname = 'visit_' + type(node).__name__  
        meth = getattr(self, methname, None)  
        if meth is None:  
            meth = self.generic_visit  
        return meth(node)  
  
    def generic_visit(self, node):  
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

Чтобы использовать этот класс, программист наследует от него и реализует различные методы формы `visit_Name()`, где вместо Name подставляется тип узла. Например, если вы хотите выполнить выражение, вы можете написать это так:

```
class Evaluator(NodeVisitor):  
    def visit_Number(self, node):  
        return node.value  
  
    def visit_Add(self, node):  
        return self.visit(node.left) + self.visit(node.right)  
  
    def visit_Sub(self, node):  
        return self.visit(node.left) - self.visit(node.right)  
  
    def visit_Mul(self, node):  
        return self.visit(node.left) * self.visit(node.right)  
  
    def visit_Div(self, node):  
        return self.visit(node.left) / self.visit(node.right)  
  
    def visit_Negate(self, node):  
        return -node.operand
```

Вот пример того, как вы можете использовать этот класс с ранее сгенерированным нами выражением:

```
>>> e = Evaluator()  
>>> e.visit(t4)  
0.6  
>>>
```

В качестве совершенно другого примера приведём класс, который

транслирует выражение в операции на простой стековой машине:

```
class StackCode(NodeVisitor):
    def generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return self.instructions

    def visit_Number(self, node):
        self.instructions.append(( 'PUSH', node.value))

    def binop(self, node, instruction):
        self.visit(node.left)
        self.visit(node.right)
        self.instructions.append((instruction,))

    def visit_Add(self, node):
        self.binop(node, 'ADD')

    def visit_Sub(self, node):
        self.binop(node, 'SUB')

    def visit_Mul(self, node):
        self.binop(node, 'MUL')

    def visit_Div(self, node):
        self.binop(node, 'DIV')

    def unaryop(self, node, instruction):
        self.visit(node.operand)
        self.instructions.append((instruction,))

    def visit_Negate(self, node):
        self.unaryop(node, 'NEG')
```

Вот пример работы этого класса:

```
>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
 ('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
>>>
```

Обсуждение

В этом рецепте две ключевых идеи. Первая — это стратегия проектирования, при которой код, который манипулирует сложной структурой данных,

отделён от самой структуры данных. Здесь это применено так: ни один из различных классов *Node* не предоставляет никаких реализаций, которые что-то делают с данными. Вместо этого все манипуляции с данными выполняются специальными реализациями отдельного класса *NodeVisitor*. Это разделение делает код максимально общим, а не специализированным.

Вторая ключевая идея — реализация класса-посетителя как такового. В посетителе вы хотите переключаться между методами обработки в зависимости от некоторого значения, такого как тип узла. В элементарной реализации вы могли бы склониться к созданию огромного объявления *if*:

```
class NodeVisitor:  
    def visit(self, node):  
        nodetype = type(node).__name__  
        if nodetype == 'Number':  
            return self.visit_Number(node)  
        elif nodetype == 'Add':  
            return self.visit_Add(node)  
        elif nodetype == 'Sub':  
            return self.visit_Sub(node)  
        ...
```

Однако быстро станет ясно, что на самом деле вам не стоит выбирать такой подход. Помимо того, что он чрезвычайно многословен, он еще и медленно работает, а также его трудно поддерживать, если вы захотите добавить или изменить типы обрабатываемых узлов. Вместо этого лучше исполнить маленький фокус, при котором вы формируете имя метода и получаете его с помощью функции *getattr()*. Метод *generic_visit()* в показанном решении — это запасной вариант, который будет применён, если не будет найден подходящий метод обработки. В этом рецепте он возбуждает исключение, чтобы предупредить программиста о том, что встретился неожиданный тип узла.

В каждом классе-посетителе вычисления обычно вызываются рекурсивными вызовами метода *visit()*. Например:

```
class Evaluator(NodeVisitor):  
    ...  
    def visit_Add(self, node):  
        return self.visit(node.left) + self.visit(node.right)
```

Рекурсия — это то, что заставляет класс-посетитель обходить всю структуру данных целиком. Вы вызываете *visit()* до тех пор, пока не достигнете некого

конечного узла, такого как *Number* в приведённом выше примере. Точный порядок рекурсии и других операций полностью зависит от приложения.

Стоит отметить, что в этом конкретном приёме переключение на нужный метод — это также обычный способ эмуляции поведения условного выражения или выражения-переключателя (*switch*), которые можно встретить в других языках. Например, если вы пишете HTTP-фреймворк, то у вас могут получиться классы, которые выполняют похожую диспетчеризацию:

```
class HTTPHandler:  
    def handle(self, request):  
        methname = 'do_' + request.request_method  
        getattr(self, methname)(request)  
  
    def do_GET(self, request):  
        ...  
    def do_POST(self, request):  
        ...  
    def do_HEAD(self, request):  
        ...
```

Слабая сторона шаблона «Посетитель» — это привязка к рекурсии. Если вы попытаетесь применить его к глубоко вложенной структуре, есть возможность достигнуть лимита Python на рекурсию (см. `sys.getrecursionlimit()`). Чтобы обойти эту проблему, вы можете делать определённые выборы в ваших структурах данных. Например, вы можете использовать обычные списки Python вместо связанных списков, или попытаться агрегировать больше данных в каждый узел, чтобы сделать структуру менее глубоко вложенной.

Вы также можете попытаться применить нерекурсивные алгоритмы обхода на основе генераторов или итераторов, как обсуждалось в [рецепте 8.22](#).

Использование шаблона «Посетитель» очень распространено в программах, связанных с парсингом или компилированием. Интересная реализация может быть найдена в модуле `ast` Python. В дополнение к возможности обхода древовидных структур, он предоставляет вариант, который позволяет переписывать и трансформировать структуру данных по мере её обхода (то есть добавлять или удалять узлы). Больше информации об этом вы найдете в исходном коде модуля `ast`. [Рецепт 9.24](#). даёт пример использования модуля `ast` для обработки исходного кода Python.

8.22. Реализация шаблона «Посетитель» без рекурсии

Задача

Вы пишете код, который обходит глубоко вложенную структуру с использованием шаблона «Посетитель», но ломается, поскольку исчерпывает лимит на рекурсию. Вы бы хотели избавиться от рекурсии, но сохранить стиль программирования, использующий паттерн «Посетитель».

Решение

Генераторы иногда можно с умом применить для устранения рекурсии из алгоритмов обхода дерева или поиска. В [рецепте 8.21.](#) был представлен класс-посетитель. Здесь мы покажем альтернативную реализацию этого класса, которая производит вычисления совершенно иным способом — с помощью стека и генераторов:

```
import types

class Node:
    pass

class NodeVisitor:
    def visit(self, node):
        stack = [ node ]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()
        return last_result

    def _visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
```

```

    if meth is None:
        meth = self.generic_visit
    return meth(node)

def generic_visit(self, node):
    raise RuntimeError('No {} method'.format('visit_' + type(node)._

```

Если вы попробуете поработать с этим классом, то обнаружите, что он по-прежнему работает с существующим кодом, который мог использовать рекурсию. На самом деле вы можете использовать его в качестве прямой замены реализации класса-посетителя в предыдущем рецепте. Рассмотрим, например, такой код с деревьями выражений:

```

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

# Пример класса-посетителя, выполняющего выражения
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

```

```

def visit_Sub(self, node):
    return self.visit(node.left) - self.visit(node.right)

def visit_Mul(self, node):
    return self.visit(node.left) * self.visit(node.right)

def visit_Div(self, node):
    return self.visit(node.left) / self.visit(node.right)

def visit_Negate(self, node):
    return -self.visit(node.operand)

if __name__ == '__main__':
    # 1 + 2*(3-4) / 5
    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)

    # Выполнить это
    e = Evaluator()
    print(e.visit(t4)) # Выведет 0.6

```

Приведённый код работает для простых выражений. Однако реализация класса *Evaluator* использует рекурсию и может поломаться, если данные будут слишком сильно вложенными. Например:

```

>>> a = Number(0)
>>> for n in range(1, 100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
Traceback (most recent call last):
...
  File "visitor.py", line 29, in _visit
    return meth(node)
  File "visitor.py", line 67, in visit_Add
    return self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>

```

Давайте немного изменим класс *Evaluator*:

```

class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

```

```

def visit_Add(self, node):
    yield (yield node.left) + (yield node.right)

def visit_Sub(self, node):
    yield (yield node.left) - (yield node.right)

def visit_Mul(self, node):
    yield (yield node.left) * (yield node.right)

def visit_Div(self, node):
    yield (yield node.left) / (yield node.right)

def visit_Negate(self, node):
    yield - (yield node.operand)

```

Если вы попробуете новую реализацию на таком же эксперименте с рекурсией, то обнаружите, что всё работает. Это магия!

```

>>> a = Number(0)
>>> for n in range(1,100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>

```

Если вы захотите добавить собственную обработку в любой из методов, то и это у вас получится. Например:

```

class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        print('Add:', node)
        lhs = yield node.left
        print('left=', lhs)
        rhs = yield node.right
        print('right=', rhs)
        yield lhs + rhs
    ...

```

Вот пример результата:

```

>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>

```

```
left= 1
right= -0.4
0.6
>>>
```

Обсуждение

Этот рецепт отлично демонстрирует, как генераторы и корутины (сопрограммы) могут выделять безумные фокусы с потоком управления, которые часто дают огромное преимущество. Чтобы понять этот рецепт, нужно уяснить для себя несколько ключевых моментов.

Во-первых, в задачах, связанных с обходом дерева, распространенной стратегией для избежания рекурсии является написание алгоритмов на основе стека или очереди. Например, обход с поиском в глубину (depth-first) может быть полностью реализован путём помещения встречающихся узлов на стек — и снятия со стека после завершения обработки. Ядро метода `visit()`, показанного в решении, базируется именно на этой идее. Алгоритм начинает с помещения в список `stack` первого узла и заканчивает работу, когда стек опустевает. Во время выполнения стек вырастает в соответствии с глубиной обрабатываемого дерева.

Второй ключевой момент касается поведения инструкции `yield` в генераторах. Когда встречается `yield`, генератор выдаёт значение и приостанавливается. Данный рецепт использует это свойство в качестве замены рекурсии. Например, вместо написания такого рекурсивного выражения:

```
value = self.visit(node.left)
```

...вы используете такое:

```
value = yield node.left
```

«Под капотом» здесь происходит отправка узла (`node.left`) обратно в метод `visit()`. Метод `visit()` затем выполняет подходящий для этого узла метод `visitName()`. В каком-то смысле это практически полная противоположность рекурсии. Вместо рекурсивных вызовов `visit()`, продвигающих вперёд ход выполнения алгоритма, инструкция `yield` используется для временных откатов осуществляемого вычисления. `yield` — это сигнал, который говорит алгоритму, что выданный узел нужно обработать перед тем, как

продвигаться дальше.

Заключительная часть этого рецепта касается распространения результатов. При использовании генераторов вы не можете применять инструкции `return` для выдачи значений (это возбудит исключение `SyntaxError`). Поэтому инструкция `yield` берёт на себя двойную функцию для выполнения этой задачи. В этом рецепте это работает так: если значение, производимое инструкцией `yield`, не является узлом, то мы предполагаем, что это значение, которое будет распространяться в следующий шаг вычисления. В этом назначение переменной `last_return`. В типичном случае оно будет удерживать последнее значение, выданное методом-посетителем. Это значение затем будет отправлено в ранее выполнявшийся метод, где оно появится как значение, возвращенное инструкцией `yield`. Например, здесь:

```
value = yield node.left
```

...переменная `value` получает значение `last_return`, которое представляет собой результат, возвращаемый методом-посетителем, вызванным для `node.left`.

Все эти аспекты рецепта можно найти в этом фрагменте кода:

```
try:
    last = stack[-1]
    if isinstance(last, types.GeneratorType):
        stack.append(last.send(last_result))
        last_result = None
    elif isinstance(last, Node):
        stack.append(self._visit(stack.pop()))
    else:
        last_result = stack.pop()
except StopIteration:
    stack.pop()
```

Код просто смотрит на вершину стека и решает, что делать дальше. Если это генератор, тогда вызывается метод `send()` с последним результатом (если он имеется), и результат добавляется на стек для дальнейшей обработки. Значение, которое возвращает `send()`, это то же самое значение, которое было передано в инструкцию `yield`. Поэтому в такой инструкции, как `yield node.left`, экземпляр класса `Node` `node.left` возвращается `send()` и помещается на вершину стека.

Если на вершине стека лежит экземпляр `Node`, он замещается результатом

вызыва подходящего для этого узла метода-посетителя. Здесь мы избавляемся от рекурсии. Вместо того, чтобы различные методы-посетители напрямую рекурсивно вызывали *visit()*, он выполняется тут. Пока методы используют *yield*, всё будет работать.

И, наконец, если на вершине стека что-то другое, то предполагается, что это какое-то возвращённое значение. Оно выталкивается со стека и помещается в *last_result*. Если следующий элемент стека — это генератор, тогда он посыпается в качестве возвращаемого значения для *yield*. Стоит отметить, что конечное возвращаемое значение *visit()* также присваивается *last_result*. Это заставляет данный рецепт работать с традиционной рекурсивной реализацией. Если генераторы не были использованы, это значение просто хранит значение, переданное какой-либо из инструкций *return()*, использованных в коде.

Потенциальная проблема этого рецепта касается различия между выдачей значений-экземпляров *Node* и значений, не являющихся экземпляром *Node*. В этой реализации все экземпляры *Node* обходятся автоматически. Это означает, что вы не можете использовать *Node* как возвращаемое значение, которое будет распространяться дальше. На практике это может и не быть важным. Однако если это всё же важно, вам может потребоваться немного адаптировать алгоритм. Например, можно добавить ещё один класс:

```
class Visit:
    def __init__(self, node):
        self.node = node

class NodeVisitor:
    def visit(self, node):
        stack = [ Visit(node) ]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Visit):
                    stack.append(self._visit(stack.pop().node))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()
        return last_result
```

```
def __visit(self, node):
    methname = 'visit_' + type(node).__name__
    meth = getattr(self, methname, None)
    if meth is None:
        meth = self.generic_visit
    return meth(node)

def generic_visit(self, node):
    raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))
```

При такой реализации различные методы-посетители будут выглядеть вот так:

```
class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        yield (yield Visit(node.left)) + (yield Visit(node.right))

    def visit_Sub(self, node):
        yield (yield Visit(node.left)) - (yield Visit(node.right))
    ...
    ...
```

Посмотрев на этот рецепт, вы можете задуматься о том, чтобы поискать решение, не использующее *yield*. Однако это может привести к тому, что у вас получится код, которому придётся разбираться с теми же проблемами, которые мы обсудили здесь. Например, чтобы устранить рекурсию, вам потребуется поддерживать стек. Вы также должны будете реализовать некую схему для управления обходом и вызова различной логики, связанной с посещениями. Без генераторов всё это закончится очень сложными манипуляциями со стеком, функциями обратного вызова (коллбэками) и прочими костылями. Собственно, главное преимущество использования *yield* как раз и заключается в том, что вы можете написать код без рекурсии в элегантном стиле, практически идентичном рекурсивной реализации.

8.23. Управление памятью в циклических структурах данных

Задача

Ваша программа создает структуры данных с циклами (например, деревья, графы, паттерны проектирования типа «Наблюдатель» и т.п.), и у вас

проблемы с управлением памятью.

Решение

Простой пример циклической структуры данных — это дерево, в котором родитель указывает на потомков, а потомки указывают на родителя. При работе с такой структурой вы должны задуматься над тем, чтобы сделать слабым один из типов ссылок, применив библиотеку `weakref`. Например:

```
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    # Свойство, которое управляет родителем с помощью слабой ссылки
    @property
    def parent(self):
        return self._parent if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

Эта реализация позволяет удалить родителя без лишнего шума. Например:

```
>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print(c1.parent)
Node('parent')
>>> del root
>>> print(c1.parent)
None
>>>
```

Обсуждение

Циклические структуры данных — это сложный аспект Python, который требует тщательного изучения, поскольку обычные правила сборки мусора к ним часто неприменимы. Например, рассмотрите такой код:

```
# Класс, созданный, чтобы проиллюстрировать то, что будет при удалении
class Data:
    def __del__(self):
        print('Data.__del__')

# Класс-узел, в котором есть цикл
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        child.parent = self
```

Теперь поэкспериментируем с этим кодом, чтобы обнаружить некоторые тонкости сборки мусора:

```
>>> a = Data()
>>> del a          # Сразу же удаляется
Data.__del__
>>> a = Node()
>>> del a          # Сразу же удаляется
Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del a          # Не удаляется (и сообщение не выводится)
>>>
```

Как вы можете видеть, объекты немедленно удаляются — за исключением последнего случая, когда в объекте присутствует цикл. Причина в том, что сборщик мусора Python построен на принципе простого подсчёта ссылок. Когда счётчик ссылок объекта достигает 0, объект сразу же удаляется. В случае циклических структур данных это, однако, никогда не произойдёт. В последней части примера узел-родитель и узел-потомок ссылаются друг на друга, что поддерживает счётчик ссылок ненулевым.

Чтобы разобраться с циклами, периодически запускается отдельный специальный сборщик мусора. Однако в качестве общего правила стоит учитывать, что вы не знаете, когда именно он запустится. Следовательно, вы

не знаете, когда циклические структуры данных будут удалены. При необходимости вы можете принудительно запустить сборку мусора, но это не лучший выход:

```
>>> import gc  
>>> gc.collect()      # Принудительный запуск  
Data.__del__  
Data.__del__  
>>>
```

Ещё более неприятная проблема возникает, если объекты, вовлечённые в цикл, определяют собственный метод `__del__()`. Предположим, что у нас есть вот такой код:

```
# Класс, созданный, чтобы проиллюстрировать то, что будет при удалении  
class Data:  
    def __del__(self):  
        print('Data.__del__')  
  
# Класс-узел, использующий цикл  
class Node:  
    def __init__(self):  
        self.data = Data()  
        self.parent = None  
        self.children = []  
  
    # НИКОГДА ТАК НЕ ДЕЛАЙТЕ.  
    # ПОКАЗАНО ТОЛЬКО ДЛЯ ДЕМОНСТРАЦИИ ПАТОЛОГИЧЕСКОГО ПОВЕДЕНИЯ  
    def __del__(self):  
        del self.data  
        del self.parent  
        del self.children  
  
    def add_child(self, child):  
        self.children.append(child)  
        child.parent = self
```

В этом случае структура данных никогда не будет удалена сборщиком мусора, и ваша программа вызовет утечку памяти! Если вы попробуете применять этот код, то увидите, что сообщение `Data.__del__` никогда не появляется — даже при принудительном запуске сборки мусора:

```
>>> a = Node()  
>>> a.add_child(Node())  
>>> del a                  # Нет сообщения (не собрано)
```

```
>>> import gc  
>>> gc.collect()          # Нет сообщения (не собрано)
```

Слабые ссылки решают эту проблему путем устранения ссылочных циклов. Слабая ссылка — это указатель на объект, который не увеличивает его счётчик ссылок. Создавать слабые ссылки можно с помощью библиотеки `weakref`:

```
>>> import weakref  
>>> a = Node()  
>>> a_ref = weakref.ref(a)  
>>> a_ref  
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>  
>>>
```

Чтобы разрешить (`dereference`) слабую ссылку, вы должны вызвать ее как функцию. Если объект всё ещё существует, он будет возвращён. В противном случае будет возвращено значение `None`. Поскольку счётчик ссылок изначального объекта не увеличивался, он может быть удалён обычным способом. Например:

```
>>> print(a_ref())  
<__main__.Node object at 0x1005c5410>  
>>> del a  
Data.__del__  
>>> print(a_ref())  
None  
>>>
```

Если вы будете использовать слабые ссылки, как показано в решении, то вы обнаружите, что ссылочные циклы не возникают, и сборка мусора происходит сразу же после того, как узел перестает использоваться. См. рецепт 8.25., чтобы получить ещё один пример использования слабых ссылок.

8.24. Заставляем классы поддерживать операции сравнения

Задача

Вы бы хотели сравнивать экземпляры вашего класса, используя обычные

операторы сравнения (т.е., `>=`, `!=`, `<=` и т.д.), но без написания большого количества специальных методов.

Решение

Классы Python могут поддерживать сравнение путём реализации специального метода для каждого из операторов сравнения. Например, чтобы обеспечить поддержку оператора `>=`, вам нужно определить в классах метод `__ge__()`. Хотя определение одного метода обычно не является проблемой, реализация методов для всех возможных операторов сравнения может быстро надоест.

Декоратор `functools.total_ordering` может помочь упростить этот процесс. Вы декорируете им класс и определяете метод `__eq__()`, а также один из методов сравнения (`__lt__()`, `__le__()`, `__gt__()` или `__ge__()`). Затем декоратор заполнит другие методы за вас.

В качестве примера давайте построим несколько домов и добавим в них несколько комнат, а затем выполним сравнения:

```
        self.style)

def __eq__(self, other):
    return self.living_space_footage == other.living_space_footage

def __lt__(self, other):
    return self.living_space_footage < other.living_space_footage
```

Здесь класс `House` был декорирован с помощью `@total_ordering`. Определения `__eq__()` и `__lt__()` предоставлены для сравнения домов на основе общего метража их комнат. Это минимальное определение — всё, что требуется, чтобы заставить работать другие операции сравнения. Например:

```
# Построим несколько домов и добавим в них комнаты
h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))

h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))

h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))

houses = [h1, h2, h3]

print('Is h1 bigger than h2?', h1 > h2)
# Выводит True

print('Is h2 smaller than h3?', h2 < h3)
# Выводит True

print('Is h2 greater than or equal to h1?', h2 >= h1)
# Выводит False

print('Which one is biggest?', max(houses))
# Выводит 'h3: 1101-square-foot Split

print('Which is smallest?', min(houses))
# Выводит 'h2: 846-square-foot Ranch'
```

Обсуждение

Если вы написали код, который заставляет класс поддерживать все основные операторы сравнения, тогда *total_ordering*, вероятно, уже не кажется вам такой уж магией: он буквально определяет отображение каждого метода поддержки сравнений на все другие, которые потребуются. Так что если вы определили `__lt__()` в вашем классе, как показано в решении, то этот метод используется для построения всех других операторов сравнения. Он просто заполняет класс методами:

```
class House:
    def __eq__(self, other):
        ...
    def __lt__(self, other):
        ...

# Методы, созданные @total_ordering
__le__ = lambda self, other: self < other or self == other
__gt__ = lambda self, other: not (self < other or self == other)
__ge__ = lambda self, other: not (self < other)
__ne__ = lambda self, other: not self == other
```

Несложно, конечно, написать такие методы вручную, но *@total_ordering* делает всё наверняка, ни о чём не забывая.

8.25. Создание закэшированных экземпляров

Задача

При создании экземпляров класса вы хотите возвращать закэшированную ссылку на предыдущий экземпляр, созданный с теми же аргументами (если они есть).

Решение

Задача, которую решает этот рецепт, иногда возникает, когда хотите убедиться, что создается только один экземпляр класса для некого набора аргументов. В качестве практического примера можно привести поведение библиотек — таких, как модуль *logging*, который создает только один

экземпляр логгера с неким конкретным именем. Например:

```
>>> import logging
>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is b
False
>>> c = logging.getLogger('foo')
>>> a is c
True
>>>
```

Чтобы реализовать такое поведение, вы должны использовать фабричную функцию, отделённую от самого класса. Например:

```
# Опрашиваемый класс
class Spam:
    def __init__(self, name):
        self.name = name

# Поддержка кэширования
import weakref
_spam_cache = weakref.WeakValueDictionary()

def get_spam(name):
    if name not in _spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else:
        s = _spam_cache[name]
    return s
```

Если вы используете эту реализацию, то обнаружите, что она ведёт себя так же, как было показано ранее:

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is b
False
>>> c = get_spam('foo')
>>> a is c
True
>>>
```

Обсуждение

Написание специальной фабричной функции часто является простым подходом для изменения обычных правил создания экземпляра. Но есть ли более элегантное решение?

Например, вы можете задуматься о переопределении метода `__new__()` в классе:

```
# Замечание: этот код не полностью рабочий
import weakref
class Spam:
    _spam_cache = weakref.WeakValueDictionary()
    def __new__(cls, name):
        if name in cls._spam_cache:
            return cls._spam_cache[name]
        else:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return self

    def __init__(self, name):
        print('Initializing Spam')
        self.name = name
```

На первый взгляд кажется, что этот код может заработать. Однако есть большая проблема: метод `__init__()` вызывается всегда — без оглядки на то, закэширован ли экземпляр. Например:

```
>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is t
True
>>>
```

Такое поведение, вероятно, нежелательно. Чтобы решить задачу кэширования без переинициализации, вам нужно попробовать слегка отличающийся подход.

Использование слабых ссылок в этом рецепте служит важной цели, связанной со сборкой мусора, что было описано в [рецепте 8.23](#). Логика такова: вы обычно хотите держать экземпляры в кэше только до тех пор, пока они используются где-то в программе. Экземпляр `WeakValueDictionary` содержит только элементы, которые существуют где-то ещё. Ключи словаря исчезают, когда экземпляры выходят из употребления. Смотрите:

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del a
>>> del c
>>> list(_spam_cache)
['bar']
>>> del b
>>> list(_spam_cache)
[]
>>>
```

Для большого количества программ будет достаточно минимального кода, показанного в этом рецепте. Однако есть несколько более продвинутых реализаций этого приёма.

При использовании этого рецепта сразу же возникает опасение по поводу использования глобальных переменных и отделённости фабричной функции от определения класса. Хороший способ подчистить эти недостатки — это поместить кэширующий код в отдельный управляющий класс, а затем склеить всё вместе:

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()
    def get_spam(self, name):
        if name not in self._cache:
            s = Spam(name)
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s

    def clear(self):
        self._cache.clear()

class Spam:
    manager = CachedSpamManager()
    def __init__(self, name):
        self.name = name

    def get_spam(name):
        return Spam.manager.get_spam(name)
```

Преимущество такого подхода в его гибкости. Например, могут быть реализованы различные схемы управления (как отдельные классы) и прикреплены к классу *Spam* в качестве замены реализации кэширования по умолчанию. Никакой другой код (например, *get_spam*) не нужно будет менять, чтобы всё продолжило работать.

Ещё один тонкий момент проектирования в этом случае заключается в том, открывать ли пользователям определение класса. Если вы ничего не предпримете, пользователь сможет легко создавать экземпляры в обход механизма кэширования:

```
>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is b
False
>>>
```

Если для вас важно это предотвратить, то это возможно. Например, вы можете дать классу имя, начинающееся с нижнего подчёркивания, такое как *_Spam*. Это сообщит пользователю о нежелательности прямого доступа.

В качестве альтернативы вы можете оставить пользователям ещё более сильный намёк о том, что они не должны создавать экземпляры *Spam*: вы можете заставить метод *__init__()* возбуждать исключение и использовать метод класса в качестве альтернативного конструктора:

```
class Spam:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name
```

Далее вы меняете кэширующий код, чтобы использовать *Spam._new()* для создания экземпляров вместо обычного вызова *Spam()*. Например:

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()
```

```
def get_spam(self, name):
    if name not in self._cache:
        s = Spam._new(name)           # Изменённое создание
        self._cache[name] = s
    else:
        s = self._cache[name]
    return s
```

Хотя есть и более экстремальные меры, которые можно предпринять для скрытия класса *Spam*, но, вероятно, лучше не зацикливаться на этой проблеме. Нижнего подчёркивания в имени или определения конструктора как метода класса обычно достаточно, чтобы программисты поняли намёк.

Кэширование и другие паттерны создания часто могут быть реализованы в более элегантной (хотя и более продвинутой) манере — через использование метаклассов. См. рецепт 9.13.

9. Метапрограммирование

Одна из самых важных мантр разработки программного обеспечения — «не повторяйтесь» (DRY, dont' repeat yourself). Поэтому каждый раз, когда вы сталкиваетесь с кодом с большим количеством повторений (или копипастом кусков кода), поиск более элегантного решения очень часто окупает себя. В Python такие задачи часто решаются путем обращения к метапрограммированию. Если вкратце, то метапрограммирование заключается в создании функций и классов, чьей главной задачей является управление кодом (то есть модификация, генерация или обёртывание существующего кода). Главные возможности — декораторы функций, декораторы классов и метаклассы. Однако в картину нужно добавить и другие полезные темы, включая сигнатуры объектов, выполнение кода с помощью `exec()` и интроспекция функций и классов. Главная цель этой главы — исследовать различные приёмы метапрограммирования и дать примеры того, как их можно использовать для подстройки поведения Python под ваши нужды.

9.1. Создание обёртки для функции

Задача

Вы хотите обернуть функцию слоем, добавляющим дополнительную логику (например, логирование, профилирование и т.п.)

Решение

Если вам потребуется обернуть функцию дополнительным кодом, определите функцию-декоратор. Например:

```
import time
from functools import wraps

def timethis(func):
    '''

    Декоратор, который выводит время выполнения.

    '''

    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

Вот пример использования декоратора:

```
>>> @timethis
... def countdown(n):
...
...
...     Counts down
...
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>
```

Обсуждение

Декоратор – это функция, которая принимает функцию на вход и возвращает новую функцию на выходе. Когда вы пишете такой код:

```
@timethis
def countdown(n):
    ...
```

...это равноценно такой последовательности шагов:

```
def countdown(n):
    ...
countdown = timethis(countdown)
```

Встроенные декораторы `@staticmethod`, `@classmethod` и `@property` работают так же. Например, два этих фрагмента эквивалентны:

```
class A:
    @classmethod
    def method(cls):
        pass

class B:
    # Эквивалентное определение метода класса
    def method(cls):
        pass
method = classmethod(method)
```

Код внутри декоратора обычно создаёт новую функцию, которая принимает любые аргументы через `*args` и `**kwargs`, как в функции `wrapper()` в этом рецепте. Внутри этой функции вы помещаете вызов изначальной входящей функции и возвращаете её результат. Однако вы также добавляете дополнительный код по желанию (например, профилирующий). Созданная функция `wrapper` возвращается в результате и занимает место изначальной функции.

Чрезвычайно важно подчеркнуть, что декораторы в общем случае не изменяют сигнатуру вызова или возвращаемое значение декорируемой функции. Использование `*args` и `**kwargs` здесь позволяет убедиться, что могут быть приняты любые входные аргументы. Возвращаемое значение декоратора практически всегда будет результатом вызова `func(*args, **kwargs)`, где `func` — это изначальная недекорированная функция.

При первой встрече с декораторами обычно начинают с простых примеров, вроде показанного выше. Однако если вы будете писать декораторы в реальной работе, то придётся помнить о нескольких тонкостях. Например, об использовании декоратора `@wraps(func)` в показанном решении легко забыть,

но именно оно обеспечивает сохранение метаданных функции (описано в следующем рецепте). Следующие несколько рецептов касаются тонкостей, связанных с написанием декораторов.

9.2. Сохранение метаданных функции при написании декораторов

Задача

Вы написали декоратор, но когда вы применяете его к функции, теряются важные метаданные — такие как имя, строка документации (doc string), аннотации и сигнатура вызова.

Решение

При определении декоратора не забывайте применить декоратор `@wraps` из библиотеки `functools` к функции-обёртке. Например:

```
import time
from functools import wraps

def timethis(func):
    """
    Декоратор, который показывает время выполнения.
    """

    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result

    return wrapper
```

Вот пример использования декоратора и исследования метаданных функции:

```
>>> @timethis
... def countdown(n:int):
...     ...
...     ...
...         Counts down
...         ...
...         while n > 0:
...             n -= 1
```

```
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown.__name__
'countdown'
>>> countdown.__doc__
'\n\tCounts down\n\t'
>>> countdown.__annotations__
{'n': <class 'int'>}
>>>
```

Обсуждение

Копирование метаданных декоратора — это важный аспект. Если вы забудете применить `@wraps`, то обнаружите, что декорированная функция потеряла различную полезную информацию. Например, если декоратор `@wraps` в последнем примере был бы опущен, метаданные выглядели бы так:

```
>>> countdown.__name__
'wrapper'
>>> countdown.__doc__
>>> countdown.__annotations__
{}
>>>
```

Важная особенность декоратора `@wraps` заключается в том, что он делает обёрнутую функцию доступной в атрибуте `__wrapped__`. Например, если вы хотите напрямую обратиться к обёрнутой функции, вы можете сделать так:

```
>>> countdown.__wrapped__(100000)
>>>
```

Наличие атрибута `__wrapped__` также позволяет декорированным функциям правильно показывать сигнатуры обёрнутых функций. Например:

```
>>> from inspect import signature
>>> print(signature(countdown))
(n:int)
>>>
```

Иногда возникает вопрос о том, как написать декоратор, который прямо копирует изначальную сигнатуру вызова обёрнутой функции (в противоположность подходу с использованием `*args` и `**kwargs`). В общем

случае этого трудно будет добиться без некоторой акробатики с генератором строк кода и функцией `exec()`. Если честно, обычно лучше использовать `@wraps` и полагаться на тот факт, что сигнтура изначальной функции может быть получена через атрибут `__wrapped__`. См. [рецепт 9.16.](#), где приведены дополнительные сведения о сигнтурах.

9.3. Снятие («разворачивание») декоратора

Задача

К функции был применён декоратор, но вы хотите «отменить» это, чтобы получить доступ к изначальной необёрнутой функции.

Решение

Предполагая, что декоратор был реализован с правильным применением `@wraps` (см. [рецепт 9.2.](#)), вы обычно можете получить доступ к оригинальной функции через обращение к атрибуту `__wrapped__`. Например:

```
>>> @somedecorator
>>> def add(x, y):
...     return x + y
...
>>> orig_add = add.__wrapped__
>>> orig_add(3, 4)
7
>>>
```

Обсуждение

Получение прямого доступа к необёрнутой функции в обход декоратора может быть полезно для целей отладки, интроспекции и прочих операций с функциями. Однако этот рецепт работает только в том случае, если реализация декоратора правильно копирует метаданные, используя `@wraps` из модуля `functools` или напрямую устанавливая атрибут `__wrapped__`.

Если к функции было применено несколько декораторов, поведение при доступе к `__wrapped__` в настоящее время не определено, и таких обращений

нужно избегать. В Python 3.3. обращение проходит сквозь все слои.

Предположим, например, что у вас есть вот такой код:

```
def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def add(x, y):
    return x + y
```

Вот что произойдёт, если вы вызовете декорированную функцию и изначальную функцию через `__wrapped__`:

```
>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
5
>>>
```

Однако такое поведение было зарегистрировано как ошибочное (см. <http://bugs.python.org/issue17482>) и может быть изменено на доступ по правильной цепочке декораторов в будущих релизах. (**Прим. пер.: В релизе Python 3.4. баг был закрыт**).

И последнее: имейте в виду, что не все декораторы используют `@wraps` и поэтому могут и не работать так, как описано. В частности, встроенные декораторы `@staticmethod` и `@classmethod` создают объекты дескриптора, которые не следуют этим соглашениям (вместо этого они сохраняют изначальную функцию в атрибуте `__func__`).

9.4. Определение декоратора, принимающего аргументы

Задача

Вы хотите создать функцию-декоратор, которая принимала бы аргументы.

Решение

Давайте покажем процесс приёма аргументов на примере. Предположим, вы хотите написать декоратор, который добавляет к функции логирование, и при этом позволяет пользователю указать уровень логирования и прочие параметры через аргументы. Вот как вы можете это сделать:

```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    """
    Добавляет логирование в функцию. level – это уровень логирования,
    name – это название логгера, message – это сообщение в лог. Если
    name и message не определены, они будут дефолтными от имени функции
    и её модуля.
    """

    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Пример использования
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

На первый взгляд реализация выглядит нетривиальной, но идея относительно проста. Самая внешняя функция *logged()* принимает желаемые аргументы и просто делает их доступными внутренним функциям декоратора. Внутренняя функция *decorate()* принимает функцию и помещает ее в обёртку обычным образом. Ключевой момент в том, что обёртка может использовать аргументы, переданные в *logged()*.

Обсуждение

Создавать декораторы, принимающие аргументы, довольно сложно, поскольку они используют «подкапотную» последовательность вызовов. Говоря конкретно, если у вас есть вот такой код:

```
@decorator(x, y, z)
def func(a, b):
    pass
```

...то процесс декорирования будет идти так:

```
def func(a, b):
    pass

func = decorator(x, y, z)(func)
```

Обратите внимание, что результат вызова *decorator(x, y, z)* должен быть вызываемым объектом, который, в свою очередь, принимает функцию и оборачивает её. В [рецепте 9.7.](#) приведён еще один пример декоратора, принимающего аргументы.

9.5. Определение декоратора с настраиваемыми пользователем атрибутами

Задача

Вы хотите написать функцию-декоратор, которая обёртывает функцию, но имеет настраиваемые пользователем атрибуты, которые могут быть использованы для управления поведением декоратора во время выполнения (в рантайме).

Решение

Это решение расширяет предыдущий рецепт путём введения функций доступа (акессоров), которые меняют внутренние переменные через объявление переменных с *nonlocal*. Функции доступа затем прикрепляются к функции-обёртке как атрибуты функции.

```
from functools import wraps, partial
import logging

# Вспомогательный декоратор для прикрепления
# к функции в качестве атрибута obj
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    """
    Добавляет логирование в функцию. level – это уровень логирования,
    name – это название логгера, message – это сообщение в лог. Если
    name и message не определены, они будут дефолтными от имени функции
    и её модуля.
    """

    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)

        # Прикрепляем функции-сеттеры
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper
    return decorate
```

```
# Пример использования
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

Вот интерактивный сеанс, который демонстрирует изменения различных атрибутов после определения:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:_main_:add
5

>>> # Изменение сообщения в лог
>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:_main_:Add called
5

>>> # Изменение уровня логирования
>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:_main_:Add called
5
>>>
```

Обсуждение

Ключевой момент этого рецепта заключается в функциях доступа (т.е., `set_message()` и `set_level()`), которые прикрепляются к обёртке в качестве атрибутов. Каждая из этих функций доступа (аксессоров) позволяет изменять внутренние параметры путём использования присвоений через `nonlocal`.

Классная возможность этого рецепта в том, что функции доступа будут распространяться через несколько уровней декорирования (если все ваши декораторы используют `@functools.wraps`). Предположим, например, что вы ввели дополнительный декоратор, такой как `@timethis` из [рецепта 9.2.](#), и написали вот такой код:

```
@timethis
```

```
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1
```

Вы обнаружите, что методы доступа всё еще работают:

```
>>> countdown(10000000)
DEBUG:_main_:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:_main_:Counting down to zero
countdown 0.8225970268249512
>>>
```

Вы также обнаружите, что они работают именно так, как если бы декораторы были указаны в обратном порядке:

```
@logged(logging.DEBUG)
@timethis
def countdown(n):
    while n > 0:
        n -= 1
```

Хотя это не показано, функции доступа, возвращающие различные настройки, могут быть легко определены путём добавления вот такого кода:

```
...
@attach_wrapper(wrapper)
def get_level():
    return level

# Альтернатива
wrapper.get_level = lambda: level
...
```

Чрезвычайно тонкий аспект этого рецепта заключается в самом выборе использования функций доступа. Вы могли бы, например, рассмотреть альтернативное решение, которое полностью основано на прямом доступе к атрибутам функции:

```
...
@wraps(func)
```

```
def wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return func(*args, **kwargs)

# Прикрепление настраиваемых атрибутов
wrapper.level = level
wrapper.logmsg = logmsg
wrapper.log = log
...
```

Этот подход будет работать до определённой точки, но только при использовании с декоратором верхнего уровня. Если у вас будет другой декоратор, добавленный поверх (такой как `@timethis` в этом примере), он заслонит лежащие ниже атрибуты и сделает их недоступными для изменения. Использование функций доступа позволяет обойти это ограничение.

И последнее: показанное в этом рецепте решение может стать альтернативой определения декораторов как классов, что показано в [рецепте 9.9](#).

9.6. Определение декоратора, принимающего необязательный аргумент

Задача

Вы хотели бы написать один декоратор, который можно было бы использовать и без аргументов — `@decorator`, и с необязательными аргументами — `@decorator(x, y, z)`. Однако вы не видите простого пути сделать это из-за различий в условиях вызова простых декораторов и декораторов, принимающих аргументы.

Решение

Вот вариант логирующего кода, показанного в [рецепте 9.5.](#), который определяет такой декоратор:

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)
```

```

logname = name if name else func.__module__
log = logging.getLogger(logname)
logmsg = message if message else func.__name__

@wraps(func)
def wrapper(*args, **kwargs):
    log.log(level, logmsg)
    return func(*args, **kwargs)
return wrapper

# Пример использования
@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')

```

Как вы можете видеть в этом примере, декоратор может быть использован как в простой форме (`@logged`), так и с необязательными аргументами (`@logged(level=logging.CRITICAL, name='example')`).

Обсуждение

Задача, которую решает этот рецепт, на самом деле относится к единообразию, консистентности в программировании. При использовании декораторов большинство программистов обычно применяют их либо без аргументов, либо с аргументами, как показано в примере. С технической точки зрения декоратор с необязательными аргументами может быть использован так:

```

@logged()
def add(x, y):
    return x+y

```

Однако это не особенно распространено, и может привести к ошибкам, если программисты будут забывать добавлять дополнительные скобки. А этот рецепт просто заставляет декоратор одинаково работать и с дополнительными скобками, и без.

Чтобы понять принцип работы кода, вы должны чётко понимать то, как декораторы применяются к функциям, а также условия их вызова. Для простого декоратора, такого как этот:

```
# Пример использования
@logged
def add(x, y):
    return x + y
```

...последовательность вызова будет такой:

```
def add(x, y):
    return x + y
add = logged(add)
```

В этом случае обёртываемая функция просто передается в *logged* первым аргументом. Поэтому в решении первый аргумент *logged()* — это обёртываемая функция. Все остальные аргументы должны иметь значения по умолчанию.

Для декоратора, принимающего аргументы, такого как этот:

```
@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

...последовательность вызова будет такой:

```
def spam():
    print('Spam!')
spam = logged(level=logging.CRITICAL, name='example')(spam)
```

При первичном вызове *logged()* обёртываемая функция не передается. Так что в декораторе она должна быть необязательной. Это, в свою очередь, заставляет другие аргументы быть именованными. Более того, когда аргументы переданы, декоратор должен вернуть функцию, которая принимает функцию и оборачивает её (см. [рецепт 9.5.](#)) Чтобы сделать это, в решении используется хитрый трюк с *functools.partial*. Если точнее, он просто возвращает частично применённую версию себя, где все аргументы зафиксированы — за исключением обёртываемой функции. См. [рецепт 7.8.](#), чтобы узнать больше об использовании *partial()*.

9.7. Принудительная проверка типов в функции с использованием декоратора

Задача

Вы хотите иметь возможность включить принудительную проверку типов аргументов функции.

Решение

Перед тем, как показывать код решения, напомним, что цель этого рецепта — получить средства для принудительной проверки правильности типов входных аргументов функции. Вот короткий пример, который иллюстрирует идею:

```
>>> @typeassert(int, int)
... def add(x, y):
...     return x + y
...
>>>
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>
```

Теперь приведём реализацию декоратора `@typeassert`:

```
from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        # Если мы в оптимизированном режиме, отключаем проверку типов
        if not __debug__:
            return func

        # Отображаем имена аргументов функции на предоставленные типы
        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Принудительно проверяем типы
            # предоставленных аргументов асертами
            ...

            return func(bound_values)

        return wrapper

    return decorate
```

```
for name, value in bound_values.arguments.items():
    if name in bound_types:
        if not isinstance(value, bound_types[name]):
            raise TypeError(
                'Argument {} must be {}'.format(name, bound_
)
    return func(*args, **kwargs)
return wrapper
return decorate
```

Вы обнаружите, что этот декоратор достаточно гибок и позволяет указать типы для всех (или для подмножества) аргументов функции. Более того, типы могут быть указаны позиционно или с помощью именованных аргументов. Вот пример:

```
>>> @typeassert(int, z=int)
... def spam(x, y, z=42):
...     print(x, y, z)
...
>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>
```

Обсуждение

В этом рецепте приведён пример продвинутого декоратора, который вводит несколько важных и полезных концепций.

Во-первых, одна из особенностей декораторов в том, что они применяются только один раз, во время определения функции. В некоторых случаях вы можете захотеть отключить функциональность, добавленную декоратором. Чтобы сделать это, просто заставьте ваш декоратор вернуть необёрнутую функцию. В решении приведённый ниже фрагмент кода возвращает неизменённую функцию, если значение глобальной переменной `_debug_` установлено на `False` (как и в том случае, когда интерпретатор Python запускается в оптимизированном режиме с параметрами `-O` или `-OO`):

```
...
def decorate(func):
    # Если мы в оптимизированном режиме, отключаем проверку типов
    if not __debug__:
        return func
...

```

Следующая тонкость написания декораторов в том, что это подразумевает изучение и работу с аргументной сигнатурой оборачиваемой функции. Оптимальный инструмент для этого — функция `inspect.signature()`. Она позволяет вам извлечь информацию о сигнатуре из вызываемого объекта. Например:

```
>>> from inspect import signature
>>> def spam(x, y, z=42):
...     pass
...
>>> sig = signature(spam)
>>> print(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x10077a1b0 'z'>)])
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<_ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

В первой части нашего декоратора мы используем метод сигнатур `bind_partial()`, чтобы выполнить частичную привязку предоставленных типов к именам аргументов. Вот пример того, как это работает:

```
>>> bound_types = sig.bind_partial(int, z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

На примере этой частичной привязки вы заметите, что недостающие аргументы просто игнорируются (то есть нет привязки для аргумента `y`). Однако наиболее важная часть привязки — это создание упорядоченного

словаря `bound_types.arguments`. Этот словарь отображает имена аргументов на предоставленные значения в том же порядке, что и сигнатура функции. В случае нашего декоратора это отображение содержит ассерты (проверки) типов, которые мы будем принудительно проверять.

В функции-обёртке, созданной декоратором, используется метод `sig.bind()`. `bind()` похож на `bind_partial()`, за исключением того, что он не позволяет пропускать аргументы. Вот как это работает:

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

Используя это отображение, относительно легко обеспечить требуемые проверки:

```
>>> for name, value in bound_values.arguments.items():
...     if name in bound_types.arguments:
...         if not isinstance(value, bound_types.arguments[name]):
...             raise TypeError()
...
>>>
```

В этом решении есть тонкий аспект: ассерты (удостоверяющие проверки) не применяются к непредоставленным аргументам со значениями по умолчанию. Например, этот код работает, хотя значение `items` по умолчанию имеет «неправильный» тип:

```
>>> @typeassert(int, list)
... def bar(x, items=None):
...     if items is None:
...         items = []
...     items.append(x)
...     return items
>>> bar(2)
[2]
>>> bar(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

Последнюю точку в обсуждении этого приёма проектирования ставит такой вопрос: использовать аргументы декораторов или аннотации функций? Почему бы, например, не написать вот такой декоратор, который будет «обращать внимание» на аннотации:

```
@typeassert
def spam(x:int, y, z:int = 42):
    print(x,y,z)
```

Возможная причина не использовать аннотации в том, что к каждому аргументу функции можно прикрепить только одну аннотацию. Поэтому если аннотации используются для проверки типов, они уже не могут быть использованы ни для чего другого. Также в этом случае декоратор `@typeassert` не будет работать с функциями, которые используют аннотации для других целей. Путём использования аргументов декоратора, как показано в решении, декоратор получает более общее назначение и может быть использовать с любой функцией — даже с теми, которые используют аннотации.

Дополнительную информацию о объектах сигнатур функций можно получить в [PEP 362](#), а также в [документации модуля inspect](#). В [рецепте 9.16.](#) вы найдёте дополнительный пример.

9.8. Определение декораторов как части класса

Задача

Вы хотите определить декоратор внутри определения класса и применить его к другим функциям или методам.

Решение

Определение декоратора внутри класса выполняется как обычно, но сначала вы должны выработать способ, которым декоратор будет применяться. В частности, будет ли он применяться как метод экземпляра или как метод класса. Вот пример, который иллюстрирует разницу:

```
from functools import wraps
```

```

class A:
    # Декоратор как метод экземпляра
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    # Декоратор как метод класса
    @classmethod
    def decorator2(cls, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper

```

Вот пример того, как эти два декоратора будут применяться:

```

# Как метод экземпляра
a = A()

@a.decorator1
def spam():
    pass

# Как метод класса
@A.decorator2
def grok():
    pass

```

Если вы посмотрите внимательно, то заметите, как один из них применяется из экземпляра *a*, а другой — из класса *A*.

Обсуждение

Определение декораторов в классе на первый взгляд может показаться странным, но примеры такого подхода вы встретите даже в стандартной библиотеке. В частности, встроенный декоратор *@property* на самом деле является классом с методами *getter()*, *setter()* и *deleter()*, каждый из которых действует как декоратор. Например:

```

class Person:
    # Создание экземпляра свойства

```

```
first_name = property()

# Применение методов декоратора
@first_name.getter
def first_name(self):
    return self._first_name

@first_name.setter
def first_name(self, value):
    if not isinstance(value, str):
        raise TypeError('Expected a string')
    self._first_name = value
```

Главная причина использования такой реализации в том, что разнообразные методы-декораторы управляют состоянием связанного экземпляра *property*. Так что если у вас когда-либо возникнет задача, связанная с необходимостью создать декораторы, которые будут записывать или комбинировать данные «за кулисами», то это будет вполне разумным подходом.

Распространённая ошибка при написании декораторов в классах — путаница с использованием дополнительных аргументов *self* или *cls* в коде самого декоратора. Хотя самая внешняя функция-декоратор, такая как *decorator1()* или *decorator2()*, нуждается в аргументе *self* или *cls* (поскольку функции являются частью класса), но функция-обёртка, создаваемая внутри, в общем случае не нуждается в дополнительном аргументе. Вот почему функция *wrapper()*, создаваемая в обоих декораторах, не включает аргумент *self*. Этот аргумент может понадобиться только в ситуации, когда вам нужен доступ к частям экземпляра в обёртке. В противном случае вам не стоит волноваться по этому поводу.

И последний тонкий момент определения декораторов в классе: использование при наследовании. Предположим, например, что вы хотите применить один из декораторов, определённых в классе *A*, к методам, определённым в подклассе *B*. Чтобы сделать это, вам нужно написать такой код:

```
class B(A):
    @A.decorator2
    def bar(self):
        pass
```

Рассматриваемый декоратор должен быть определён как метод класса, и вы

должны явно использовать имя родительского класса *A* при применении. Вы не можете использовать имя типа *@B.decorator2*, потому что во время определения метода класс *B* ещё не был создан.

9.9. Определение декораторов как классов

Задача

Вы хотите оборачивать функции декоратором, но результат должен быть вызываемым объектом. Вы хотите, чтобы ваш декоратор работал и внутри, и снаружи определения класса.

Решение

Чтобы определить декоратор как экземпляр, вы должны убедиться, что в нём реализованы методы `__call__()` и `__get__()`. Например, этот код определяет класс, который обертывает функцию простым профилирующим слоем:

```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

Чтобы использовать этот класс, примените его, как обычный декоратор — снаружи или внутри класса:

```
@Profiled
def add(x, y):
```

```
return x + y

class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)
```

Вот интерактивный сеанс, который показывает, как работают эти функции:

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
>>> Spam.bar.ncalls
3
```

Обсуждение

Определение декоратора как класса не вызывает особых сложностей. Однако есть несколько тонких аспектов, которые заслуживают быть объяснёнными — особенно если вы планируете применить декоратор к методам экземпляров.

Во-первых, использование функции `functools.wraps()` здесь служит той же цели, что и в обычных декораторах, а именно копированию важных метаданных из обёрнутой функции в вызываемый экземпляр.

Во-вторых, часто забывают про метод `__get__()`, показанный в решении. Если вы опустите `__get__()` и оставите код без изменений, то при попытке вызвать декорированные методы экземпляра обнаружите странные вещи. Например:

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...
TypeError: spam() missing 1 required positional argument: 'x'
```

Причина поломки в том, что когда функции, реализующие методы, обращаются к классу, и их метод `__get__()` вызывается как часть протокола дескриптора, который описан в [рецепте 8.9](#). В этом случае назначение метода `__get__()` в создании связанного объекта метода (который поставляет аргумент `self` методу). Вот пример, который иллюстрирует лежащую в основе механику:

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

В этом рецепте метод `__get__()` нужен, чтобы убедиться, что связанные объекты методов создаются правильно. `type.MethodType()` здесь создает связанный метод вручную. Связанные методы создаются только в том случае, если экземпляр используется. Если метод запрашивает доступ к классу, аргументу `instance` функции `__get__()` присваивается значение `None`, и возвращается сам экземпляр `Profiled`. Это делает возможным извлечение атрибута его атрибута `ncalls`, что и было показано.

Если вы не хотите возиться с этим беспорядком, рассмотрите альтернативную реализацию декоратора, использующую замыкания и переменные с объявлением `nonlocal`, как показано в [рецепте 9.5](#). Например:

```
import types
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper

# Пример
@profiled
def add(x, y):
    return x + y
```

Этот пример работает практически точно так же — за исключением того, что доступ к `ncalls` теперь предоставляется через функцию, прикреплённую к функции в качестве атрибута. Например:

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```

9.10. Применение декораторов к методам класса и статическим методам

Задача

Вы хотите применить декоратор к методу класса или статическому методу.

Решение

Применение декораторов к методам класса и статическим методам выполняется обычным способом, но нужно убедиться, что ваши декораторы применяются к методам перед `@classmethod` и `@staticmethod`. Например:

```
import time
from functools import wraps

# Простой декоратор
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper

# Класс, иллюстрирующий применение декоратора к различным типам методов
class Spam:
    @timethis
    def instance_method(self, n):
```

```

    print(self, n)
    while n > 0:
        n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1

```

Получившиеся метод класса и статический метод должны работать как обычно, но с добавлением дополнительной функциональности:

```

>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>

```

Обсуждение

Если вы перепутаете порядок применения декораторов, то получите ошибку. Например, если вы сделаете так:

```

class Spam:
    ...
    @timethis
    @staticmethod
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1

```

...то статический метод вызовет ошибку:

```
>>> Spam.static_method(1000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "timethis.py", line 6, in wrapper
      start = time.time()
TypeError: 'staticmethod' object is not callable
>>>
```

Проблема в том, что `@classmethod` и `@staticmethod` на самом деле не создают объекты, которые можно вызывать напрямую. Вместо этого они создают специальные объекты дескрипторов, как описано в [рецепте 8.9](#). Так что если вы попытаетесь использовать их в качестве функций в другом декораторе, ваш декоратор упадёт с ошибкой. Убедитесь, что эти декораторы идут первыми по порядку в списке декораторов, и это решит проблему.

Есть ситуация, где этот рецепт имеет чрезвычайную важность: при определении классов и статических методов в абстрактных базовых классах, как описано в [рецепте 8.12](#). Например, если вы хотите определить метод абстрактного класса, вы можете использовать такой код:

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @classmethod
    @abstractmethod
    def method(cls):
        pass
```

В этом фрагменте порядок `@classmethod` и `@abstractmethod` имеет значение. Если вы поменяете местами эти декораторы, всё поломается.

9.11. Написание декораторов, которые добавляют аргументы обёрнутым функциям

Задача

Вы хотите написать декоратор, который добавляет дополнительный аргумент в сигнатуру вызова обёртываемой функции. Однако добавленный аргумент

не может перекрываться с существующими условиями вызова функции.

Решение

Дополнительные аргументы могут быть внедрены в сигнатуру вызова путём использования обязательных именованных аргументов. Рассмотрите такой декоратор:

```
from functools import wraps

def optional_debug(func):
    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Вот пример его работы:

```
>>> @optional_debug
... def spam(a,b,c):
...     print(a,b,c)
...
>>> spam(1,2,3)
1 2 3
>>> spam(1,2,3, debug=True)
Calling spam
1 2 3
>>>
```

Обсуждение

Добавление аргументов в сигнатуру обёрнутой функции — не самое распространённое применение декораторов. Однако это может оказаться полезным приёмом для избежания некоторых шаблонов повторения кода. Например, если у вас есть такой код:

```
def a(x, debug=False):
    if debug:
        print('Calling a')
    ...

def b(x, y, z, debug=False):
```

```
if debug:  
    print('Calling b')  
...  
  
def c(x, y, debug=False):  
    if debug:  
        print('Calling c')  
...  
...
```

...то вы можете отрефакторить его вот так:

```
@optional_debug  
def a(x):  
...  
  
@optional_debug  
def b(x, y, z):  
...  
  
@optional_debug  
def c(x, y):  
...  
...
```

Реализация этого рецепта базируется на том факте, что обязательные именованные аргументы легко добавить в функции, которые также принимают параметры `*args` и `**kwargs`. При использовании обязательного именованного аргумента он выделяется в специальный случай и удаляется из последующих вызовов, которые используют только оставшиеся позиционные и именованные аргументы.

Здесь есть хитрость, которая касается потенциального конфликта имён между добавленным аргументом и аргументами оборачиваемой функции. Например, если декоратор `@optional_debug` был применён к функции, которая уже имела аргумент `debug`, тогда всё поломается. Если это вызывает опасения, то можно добавить дополнительную проверку:

```
from functools import wraps  
import inspect  
  
def optional_debug(func):  
    if 'debug' in inspect.getargspec(func).args:  
        raise TypeError('debug argument already defined')  
  
    @wraps(func)  
    def wrapper(*args, debug=False, **kwargs):  
        if debug:  
...
```

```

    print('Calling', func.__name__)
    return func(*args, **kwargs)
return wrapper

```

Последнее уточнение касается правильного управления сигнатурами функций. Проницательный программист поймёт, что сигнатура обёрнутых функций будет неправильной. Например:

```

>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> import inspect
>>> print(inspect.signature(add))
(x, y)

```

Это может быть исправлено с помощью такой модификации:

```

from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    sig = inspect.signature(func)
    parms = list(sig.parameters.values())
    parms.append(inspect.Parameter('debug',
                                   inspect.Parameter.KEYWORD_ONLY,
                                   default=False))
    wrapper.__signature__ = sig.replace(parameters=parms)
    return wrapper

```

Теперь сигнатура обёртки будет правильно отражать присутствие аргумента *debug*. Например:

```

>>> @optional_debug
... def add(x,y):
...     return x+y
>>> print(inspect.signature(add))
(x, y, *, debug=False)

```

```
>>> add(2,3)
5
>>>
```

Обратитесь к [рецепту 9.16.](#) за сведениями о сигнатурах функций.

9.12. Использование декораторов как патчей определений классов

Задача

Вы хотите инспектировать или переписать части определения класса, чтобы изменить его поведение, но без использования наследования или метаклассов.

Решение

Для этого очень удобно использовать декоратор класса. Например, вот декоратор класса, который изменяет специальный метод `__getattribute__`, чтобы добавить логирование.

```
def log_getattribute(cls):
    # Получение изначальной реализации
    orig_getattribute = cls.__getattribute__

    # Создание нового определения
    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    # Прикрепление к классу и возврат
    cls.__getattribute__ = new_getattribute
    return cls

# Пример использования
@log_getattribute
class A:
    def __init__(self, x):
        self.x = x
    def spam(self):
        pass
```

Вот как это работает:

```
>>> a = A(42)
>>> a.x
getting: x
42
>>> a.spam()
getting: spam
>>>
```

Обсуждение

Декораторы классов часто используют как простую альтернативу более продвинутым приёмам типа миксинов (примесей) и метаклассов. Например, альтернативная реализация вышеприведённого решения может быть создана на базе наследования:

```
class LoggedGetattribute:
    def __getattribute__(self, name):
        print('getting:', name)
        return super().__getattribute__(name)

# Пример
class A(LoggedGetattribute):
    def __init__(self, x):
        self.x = x
    def spam(self):
        pass
```

Это работает, но чтобы понять такой код, нужно уделить внимание порядку разрешения методов, функции `super()` и прочим аспектам наследования, описанным в рецепте 8.7. В каком-то смысле решение с использованием декоратора класса работает намного более прямолинейно и не вводит новых зависимостей в иерархию наследования. Как оказывается, такая реализация и работает немного быстрее, поскольку не полагается на функцию `super()`.

Если вы применяете несколько декораторов к классу, порядок применения может иметь значение. Например, декоратор, который заменяет метод полностью новой реализацией, вероятно, должен быть применен перед декоратором, который просто оборачивает существующий метод дополнительной логикой.

См. рецепт 8.13., где приведён ещё один пример использования декораторов классов.

9.13. Использование метакласса для управления созданием экземпляров

Задача

Вы хотите изменить процесс создания экземпляров с целью реализовать синглтон, кэширование или другие похожие возможности.

Решение

Любой Python-разработчик знает, что если определить класс, то можно вызывать его как функцию и создавать экземпляры. Например:

```
class Spam:  
    def __init__(self, name):  
        self.name = name  
    a = Spam('Guido')  
    b = Spam('Diana')
```

Если вы хотите кастомизировать этот шаг, то можете определить метакласс и нужным вам образом заново реализовать его метод `__call__()`. Чтобы проиллюстрировать это, предположим, что вы не хотите никому позволять создавать экземпляры:

```
class NoInstances(type):  
    def __call__(self, *args, **kwargs):  
        raise TypeError("Can't instantiate directly")  
  
# Пример  
class Spam(metaclass=NoInstances):  
    @staticmethod  
    def grok(x):  
        print('Spam.grok')
```

В этом случае пользователи могут вызвать определённый статический метод, но создать экземпляр обычным путём невозможно. Например:

```
>>> Spam.grok(42)  
Spam.grok  
>>> s = Spam()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>
```

```
File "example1.py", line 7, in __call__
    raise TypeError("Can't instantiate directly")
TypeError: Can't instantiate directly
>>>
```

А теперь предположим, что вы хотите реализовать синглтон (паттерн проектирования «Одиночка») — класс, из которого можно создать только один экземпляр. Это делается относительно прямолинейно:

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
        return self.__instance
    else:
        return self.__instance

# Пример
class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')
```

В этом случае можно будет создать только один экземпляр. Например:

```
>>> a = Spam()
Creating Spam
>>> b = Spam()
>>> a is b
True
>>> c = Spam()
>>> a is c
True
>>>
```

И, наконец, предположим, что вы хотите создавать кэшированные экземпляры, как описано в **рецепте 8.25**. Вот метакласс, который это реализует:

```
import weakref

class Cached(type):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

```

self.__cache = weakref.WeakValueDictionary()

def __call__(self, *args):
    if args in self.__cache:
        return self.__cache[args]
    else:
        obj = super().__call__(*args)
        self.__cache[args] = obj
    return obj

# Пример
class Spam(metaclass=Cached):
    def __init__(self, name):
        print('Creating Spam({!r})'.format(name))
        self.name = name

```

Вот как это работает:

```

>>> a = Spam('Guido')
Creating Spam('Guido')
>>> b = Spam('Diana')
Creating Spam('Diana')
>>> c = Spam('Guido')    # Закэширован
>>> a is b
False
>>> a is c              # Возвращается закэшированное значение
True
>>>

```

Обсуждение

Использование метаклассов для реализации различных паттернов создания экземпляров часто может оказаться более элегантным решением, нежели подходы без применения метаклассов. Если, например, вы не используете метакласс, то вам может потребоваться спрятать классы за какой-то дополнительной фабричной функцией. Чтобы создать синглтон, вам будет нужен такой хак:

```

class _Spam:
    def __init__(self):
        print('Creating Spam')

    _spam_instance = None
    def Spam():
        global _spam_instance
        if _spam_instance is not None:

```

```
        return _spam_instance
    else:
        _spam_instance = _Spam()
    return _spam_instance
```

Хотя решение с метаклассами использует намного более продвинутую концепцию, получающийся код будет чище и менее «хакнутым».

См. рецепт 8.25., где приведена информация о создании кэшированных экземпляров, слабых ссылках и прочих деталях.

9.14. Захват порядка определения атрибутов класса

Задача

Вы хотите автоматически записывать порядок, в котором внутри тела класса определяются атрибуты и методы, что полезно при различных операциях (например, при сериализации, отображении в базы данных и т.п.)

Решение

Захват информации о теле определения класса легко реализуется через использование метакласса. Вот пример метакласса, который использует *OrderedDict* для захвата порядка определения дескрипторов:

```
from collections import OrderedDict

# Набор дескрипторов для различных типов
class Typed:
    _expected_type = type(None)
    def __init__(self, name=None):
        self._name = name

    def __set__(self, instance, value):
        if not isinstance(value, self._expected_type):
            raise TypeError('Expected ' + str(self._expected_type))
        instance.__dict__[self._name] = value

class Integer(Typed):
    _expected_type = int

class Float(Typed):
    _expected_type = float
```

```

    _expected_type = float

class String(Typed):
    _expected_type = str

# Метакласс, который использует OrderedDict для тела класса
class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for name, value in clsdict.items():
            if isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return OrderedDict()

```

В этом метаклассе порядок определения дескрипторов захватывается путём использования *OrderedDict* при выполнении тела класса. Получившийся порядок имён затем извлекается из словаря и сохраняется в атрибуте класса *_order*. Далее он может быть использован методами класса самыми разнообразными способами. Например, вот простой класс, который использует эту информацию о порядке для реализации метода для сериализации данных экземпляра в строчки CSV-данных:

```

class Structure(metaclass=OrderedMeta):
    def as_csv(self):
        return ','.join(str(getattr(self, name)) for name in self._order)

# Пример использования
class Stock(Structure):
    name = String()
    shares = Integer()
    price = Float()
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

Вот как работает этот класс *Stock*:

```
>>> s = Stock('GOOG', 100, 490.1)
```

```
>>> s.name
'GOOG'
>>> s.as_csv()
'GOOG,100,490.1'
>>> t = Stock('AAPL','a lot', 610.23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dupmethod.py", line 34, in __init__
TypeError: shares expects <class 'int'>
>>>
```

Обсуждение

Краеугольный камень этого рецепта — метод `__prepare__()`, который определён в метаклассе `OrderedMeta`. Этот метод немедленно вызывается при старте определения класса вместе с именем класса и базовыми классами. Он должен вернуть объект отображения для использования во время выполнения тела класса. Порядок определения легко захватить и сохранить с помощью возвращения `OrderedDict` вместо обычного словаря.

Можно дополнительно расширить эту функциональность, если вы хотите сделать свои собственные «словареподобные» объекты. Например, рассмотрите такой вариант решения, которое отвергает дублирующиеся определения:

```
from collections import OrderedDict

class NoDupOrderedDict(OrderedDict):
    def __init__(self, clsname):
        self.clsname = clsname
        super().__init__()
    def __setitem__(self, name, value):
        if name in self:
            raise TypeError('{} already defined in {}'.format(name, self))
        super().__setitem__(name, value)

class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        d['_order'] = [name for name in clsdict if name[0] != '_']
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return NoDupOrderedDict(clsname)
```

Вот что произойдет, если вы используете этот метакласс и создадите класс с дублирующимися записями:

```
>>> class A(metaclass=OrderedMeta):
...     def spam(self):
...         pass
...     def spam(self):
...         pass
...
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in A
  File "dupmethod2.py", line 25, in __setitem__
    (name, self.clsname))
TypeError: spam already defined in A
>>>
```

Последний важный аспект этого рецепта касается обращения с модифицированным словарём в методе метакласса `__new__()`. Хотя класс определён с использованием альтернативного словаря, вы всё равно должны преобразовать этот словарь в правильный экземпляр `dict` при создании конечного объекта класса. В этом назначение инструкции `d = dict(clsdict)`.

Возможность захвата порядка определения — аспект, который весьма важен для некоторых приложений. Например, в реализации объектно-реляционного отображения (ORM) классы могут быть написаны в стиле, который похож на показанный в примере:

```
class Stock(Model):
    name = String()
    shares = Integer()
    price = Float()
```

Лежащий в основе код может захотеть захватить порядок определения, чтобы отобразить объекты на кортежи или строки в таблице базы данных (похожим образом работает метод `as_csv()` в примере). Показанное решение весьма бесхитростно и чаще будет проще альтернатив (обычно это поддержание скрытых счётчиков в классах-дескрипторах).

9.15. Определение метакласса, принимающего необязательные аргументы

Задача

Вы хотите определить метакласс, который позволяет предоставлять определениям классов необязательные аргументы — это может пригодиться, например, для управления и конфигурирования обработки при создании типа.

Решение

При определении классов Python позволяет указывать метакласс с помощью именованного аргумента *metaclass* в инструкции *class*. Например, для абстрактных базовых классов это делается так:

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxsize=None):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

Однако в ваших собственных метаклассах могут быть предоставлены дополнительные именованные аргументы:

```
class Spam(metaclass=MyMeta, debug=True, synchronize=True):
    ...
```

Для поддержки таких именованных аргументов в метаклассе вам нужно убедиться, что вы определили их в методах *__prepare__()*, *__new__()* и *__init__()*, используя обязательные именованные аргументы:

```
class MyMeta(type):
    # Необязательно
    @classmethod
    def __prepare__(cls, name, bases, *, debug=False, synchronize=False)
        # Кастомная обработка
        ...
        return super().__prepare__(name, bases)

    # Требуется
```

```
def __new__(cls, name, bases, ns, *, debug=False, synchronize=False)
    # Кастомная обработка
    ...
    return super().__new__(cls, name, bases, ns)

# Требуется
def __init__(self, name, bases, ns, *, debug=False, synchronize=False)
    # Кастомная обработка
    ...
    super().__init__(name, bases, ns)
```

Обсуждение

Добавление необязательных именованных аргументов в метакласс требует, чтобы вы понимали все шаги создания класса, потому что дополнительные аргументы передаются каждому вовлечённому методу. Метод `__prepare__()` вызывается первым и используется для создания пространства имён класса перед тем, как тело любого определения в классе будет обработано. В обычном случае этот метод просто возвращает словарь или другой объект отображения. Метод `__new__()` используется для создания экземпляра получившегося объекта данного типа. Он вызывается после того, как тело класса полностью выполнено. Метод `__init__()` вызывается последним и используется для выполнения любых дополнительных шагов инициализации.

При написании метаклассов обычно определяют только один из методов `__new__()` или `__init__()`, но не оба. Однако если дополнительный именованный аргумент должен быть принят, оба метода должны предоставлены и иметь совместимые сигнатуры. Дефолтный метод `__prepare__()` принимает любой набор именованных аргументов, но игнорирует их. Вам только нужно самостоятельно определить, будут ли дополнительные аргументы как-то влиять на управление созданием пространства имён класса.

Использование обязательных именованных аргументов в этом рецепте отражает тот факт, что такие аргументы будут предоставлены по ключевым словам во время создания класса.

Использование именованных аргументов для конфигурации метакласса может быть рассмотрено в качестве альтернативы использованию переменных класса для решения той же задачи. Например:

```
class Spam(metaclass=MyMeta):
    debug = True
```

```
synchronize = True  
...
```

Преимущество предоставления таких параметров в качестве аргументов в том, что они не загрязняют пространство имён класса дополнительными именами, которые имеют отношение только к созданию класса, а не к последующему выполнению инструкций в классе. Также они доступны методу `__prepare__()`, который запускается до начала обработки любой инструкции в теле класса. Переменные класса, с другой стороны, будут доступны только в методах метакласса `__new__()` и `__init__()`.

9.16. Принудительная установка аргументной сигнатуры при использовании *args и **kwargs

Задача

Вы написали функцию или метод, который использует `*args` и `**kwargs`, чтобы обеспечить максимально общее назначение, но вы также хотели бы иметь возможность проверять передаваемые аргументы, чтобы убедиться, что они совпадают с определённой сигнатурой вызова функции.

Решение

Для любой задачи, где вы хотите манипулировать сигнатурами вызова функций, вы должны использовать связанные с сигнатурами возможности из модуля `inspect`. Особенный интерес представляют два класса — `Signature` и `Parameter`. Вот интерактивный сеанс создания сигнатуры функции:

```
>>> from inspect import Signature, Parameter  
>>> # Создаём сигнатуру для func(x, y=42, *, z=None)  
>>> parms = [ Parameter('x', Parameter.POSITIONAL_OR_KEYWORD),  
...             Parameter('y', Parameter.POSITIONAL_OR_KEYWORD, default=42),  
...             Parameter('z', Parameter.KEYWORD_ONLY, default=None) ]  
>>> sig = Signature(parms)  
>>> print(sig)  
(x, y=42, *, z=None)  
>>>
```

Когда вы получаете объект сигнатуры, вы легко можете связать его с `*args` и `**kwargs`, используя метод сигнатур `bind()`, как показано в этом простом примере:

```
>>> def func(*args, **kwargs):
...     bound_values = sig.bind(*args, **kwargs)
...     for name, value in bound_values.arguments.items():
...         print(name,value)
...
>>> # Поехали экспериментировать со значениями
>>> func(1, 2, z=3)
x 1
y 2
z 3
>>> func(1)
x 1
>>> func(1, z=3)
x 1
z 3
>>> func(y=2, x=1)
x 1
y 2
>>> func(1, 2, 3, 4)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1972, in _bind
    raise TypeError('too many positional arguments')
TypeError: too many positional arguments
>>> func(y=2)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1961, in _bind
    raise TypeError(msg) from None
TypeError: 'x' parameter lacking default value
>>> func(1, y=2, x=3)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1985, in _bind
    '{arg!r}'.format(arg=param.name))
TypeError: multiple values for argument 'x'
>>>
```

Как вы можете видеть, привязка сигнатуры к передаваемым аргументам обеспечивает принудительное выполнение всех обычных правил вызова функции, касающихся требуемых аргументов, значений по умолчанию, дубликатов и т.д.

Ниже приведён более конкретный пример принудительного использования

сигнатур функций. В этом коде базовый класс определяет `__init__()` максимально широкого назначения, но подклассы должны предоставить ожидаемую сигнатуру.

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
              for name in names]
    return Signature(parms)

class Structure:
    __signature__ = make_sig()
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Пример использования
class Stock(Structure):
    __signature__ = make_sig('name', 'shares', 'price')

class Point(Structure):
    __signature__ = make_sig('x', 'y')
```

Вот пример работы класса `Stock`:

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> s1 = Stock('ACME', 100, 490.1)
>>> s2 = Stock('ACME', 100)
Traceback (most recent call last):
...
TypeError: 'price' parameter lacking default value
>>> s3 = Stock('ACME', 100, 490.1, shares=50)
Traceback (most recent call last):
...
TypeError: multiple values for argument 'shares'
>>>
```

Обсуждение

Использование функций с `*args` и `**kwargs` очень распространено при создании библиотек общего назначения, написании декораторов или реализации прокси. Однако один из недостатков таких функций в том, что

если вы захотите реализовать собственную проверку аргументов, то всё быстро превратится в нераспутываемый клубок. В качестве примера см. **рецепт 8.11.** Использование объекта сигнатуры это упрощает.

В последнем примере этого решения имеет смысл создать объекты сигнатур через использование кастомных метаклассов. Вот альтернативная реализация, которая показывает, как это сделать:

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
              for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields', []))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Пример
class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']
```

При определении собственных сигнатур часто бывает полезно сохранить сигнатуру в специальном атрибуте `__signature__`, как показано выше. Если вы сделаете это, то код, использующий модуль `inspect` для интроспекции, увидит сигнатуру и сообщит о ней, как об условии вызова. Например:

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> print(inspect.signature(Point))
(x, y)
>>>
```

9.17. Принуждение к использованию соглашений о кодировании в классах

Задача

Ваша программа состоит из обширной иерархии классов, и вы хотели бы принудительно внедрить некоторые соглашения о кодировании (или произвести диагностику их применения), чтобы помочь сохранить чистоту кода.

Решение

Если вы хотите отслеживать определение классов, вы часто можете сделать это путём определения метакласса. Базовый метакласс обычно определяется путём наследования от `type` и переопределения метода `__new__()` или `__init__()`. Например:

```
class MyMeta(type):
    def __new__(self, clsname, bases, clsdic):
        # clsname – имя определённого класса
        # bases – кортеж базового класса
        # clsdic – словарь класса
        return super().__new__(cls, clsname, bases, clsdic)
```

Альтернативное решение, если определён `__init__()`:

```
class MyMeta(type):
    def __init__(self, clsname, bases, clsdic):
        super().__init__(clsname, bases, clsdic)
        # clsname – имя определённого класса
        # bases – кортеж базового класса
        # clsdic – словарь класса
```

Чтобы использовать метакласс, в общем случае вы могли бы внедрить его в базовый класс высшего уровня, который наследуют прочие объекты.

Например:

```
class Root(metaclass=MyMeta):
    pass

class A(Root):
    pass
```

```
class B(Root):
    pass
```

Ключевая возможность метакласса в том, что он позволяет вам исследовать содержимое класса во время определения. Внутри переопределённого метода `__init__()` вы можете свободно инспектировать словарь класса, базовые классы и так далее. Более того, после того, как для класса определён метакласс, он наследуется всеми подклассами. С помощью этого ловкий создатель фреймворка может указать метакласс для одного из классов высшего уровня в обширной иерархии и отлавливать определение всех подклассов.

В качестве конкретного, хотя и вычурного примера приведём метакласс, который отвергает любые определения классов, содержащие методы с названиями, написанными в смешанных регистрах, т.е. в camelCase (например, чтобы позлить Java-программистов):

```
class NoMixedCaseMeta(type):
    def __new__(cls, clsname, bases, clsdic):
        for name in clsdic:
            if name.lower() != name:
                raise TypeError('Bad attribute name: ' + name)
        return super().__new__(cls, clsname, bases, clsdic)

class Root(metaclass=NoMixedCaseMeta):
    pass

class A(Root):
    def foo_bar(self):    # Ok
        pass

class B(Root):
    def fooBar(self):    # TypeError
        pass
```

В качестве более продвинутого и полезного примера приведём метакласс, который проверяет определение переопределённых методов, чтобы убедиться, что они имеют такую же сигнатуру вызова, как и изначальный метод суперкласса.

```
from inspect import signature
import logging

class MatchSignaturesMeta(type):
```

```

def __init__(self, clsname, bases, clsdict):
    super().__init__(clsname, bases, clsdict)
    sup = super(self, self)
    for name, value in clsdict.items():
        if name.startswith('_') or not callable(value):
            continue
        # Получить предыдущее определение
        # (если оно есть) и сравнение сигнатур
        prev_dfn = getattr(sup, name, None)
        if prev_dfn:
            prev_sig = signature(prev_dfn)
            val_sig = signature(value)
            if prev_sig != val_sig:
                logging.warning('Signature mismatch in %s. %s != %s'
                                % (value.__qualname__, prev_sig, val_sig))

# Пример
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

# Класс с переопределёнными методами,
# но со слегка различными сигнатурами
class B(A):
    def foo(self, a, b):
        pass

    def spam(self, x, z):
        pass

```

Если вы запустите этот код, то получите такой вывод:

```

WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x,
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)

```

Такие предупреждения могут быть полезны для отлавливания сложных багов. Например, код, который полагается на передачу именованного аргумента в метод, сломается, если подкласс меняет имена аргументов.

Обсуждение

В больших объектно-ориентированных программах часто может быть полезно поместить определение классов под контроль метакласса. Метакласс может наблюдать за определением классов и использоваться для предупреждения программистов о возможных проблемах, которые могли бы пройти незамеченными (например, использование слегка несовместимых сигнатур методов).

Тут можно возразить, что такие ошибки лучше бы отлавливать инструментом анализа или с помощью IDE. Однако если вы создаете фреймворк или библиотеку, которую будут использовать другие, часто у вас нет контроля над аккуратностью подхода пользователей к разработке. Поэтому для некоторых типов приложений может иметь смысл поместить дополнительные проверки в метакласс — если такая проверка предоставит дополнительное удобство пользователям.

Выбор между переопределением `__new__()` или `__init__()` в метаклассе зависит от того, как вы хотите работать с получившимся классом. `__new__()` вызывается до создания класса и обычно используется, если метакласс хочет как-то изменить определение класса (путем изменения содержания словаря класса). Метод `__init__()` вызывается после создания класса, и он полезен, если вы хотите написать код, который работает с полностью сформированным объектом класса. В последнем примере это необходимо, поскольку он использует функцию `super()` для поиска вышестоящих в иерархии наследования определений. Это работает, только если объект класса создан и лежащий в основе порядок разрешения методов был установлен.

Последний пример также иллюстрирует использование объекта сигнатуры функции Python. Метакласс принимает каждое определение вызываемого объекта в классе, ищет предыдущее определение в иерархии (если оно есть), а затем просто сравнивает их сигнатуры вызова, используя `inspect.signature()`.

И последнее: строчка кода, которая использует `super(self, self)`, не является опечаткой. При работе с метаклассом важно понимать, что `self` — это объект класса. Так что это объявление на самом деле используется для поиска определений, размещённых выше в классовой иерархии, которые создают родителей `self`.

9.18. Программное определение классов

Задача

Вы пишете код, задача которого — создать новый объект класса. Вы подумываете о том, чтобы отправить исходный код класса в строковом формате в функцию `exec()` для выполнения, но предпочли бы более элегантное решение.

Решение

Вы можете использовать функцию `types.new_class()` для создания новых объектов классов. Вам нужно предоставить ей имя класса, кортеж с родительскими классами, именованные аргументы и функцию обратного вызова (коллбэк), который заполнит словарь класса элементами. Например:

```
# stock.py
# Пример ручного создания класса из частей

# Методы
def __init__(self, name, shares, price):
    self.name = name
    self.shares = shares
    self.price = price

def cost(self):
    return self.shares * self.price

cls_dict = {
    '__init__' : __init__,
    'cost' : cost,
}

# Создание класса
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))
Stock.__module__ = __name__
```

Это создаст обычный объект класса, который работает именно так, как ожидается:

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
```

```
4555.0
```

```
>>>
```

В решении есть тонкий аспект — присваивание `Stock.__module__` после вызова `types.new_class()`. Когда класс определён, атрибут `__module__` содержит имя модуля, в котором он был определён. Это имя используется для вывода, который производят методы типа `__repr__()`. Оно также используется различными библиотеками, такими как `pickle`. Так что если вы хотите, чтобы созданный класс был «настоящим», убедитесь, что этот атрибут установлен правильно.

Если класс, который вы хотите создать, использует метаклассы, это может быть определено в третьем аргументе, передаваемом в `types.new_class()`. Например:

```
>>> import abc
>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMeta},
...                               lambda ns: ns.update(cls_dict))
...
>>> Stock.__module__ = __name__
>>> Stock
<class '__main__.Stock'>
>>> type(Stock)
<class 'abc.ABCMeta'>
>>>
```

Третий аргумент также может содержать другие именованные аргументы. Например, такое определение класса:

```
class Spam(Base, debug=True, typecheck=False):
    ...
```

...аналогично такому вызову `new_class()`:

```
Spam = types.new_class('Spam', (Base,),
                      {'debug': True, 'typecheck': False},
                      lambda ns: ns.update(cls_dict))
```

Четвёртый аргумент `new_class()` — самый загадочный. Это функция, которая принимает на вход объект отображения, который используется для хранения пространства имён класса. Обычно это словарь, но на самом деле это объект, который возвращается методом `__prepare__()`, как описано в [рецепте 9.14](#). Эта функция должна добавить новые записи к пространству имён,

используя метод `update()` (как показано) или другие операции над объектами отображений.

Обсуждение

Умение создавать классы таким образом может быть весьма полезно в некоторых обстоятельствах. Один из наиболее знакомых разработчикам примеров — использование функции `collections.namedtuple()`. Например:

```
>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>
>>>
```

`namedtuple()` использует `exec()` вместо показанного в решении приёма. Однако есть простой вариант, который создает класс напрямую:

```
import operator
import types
import sys

def named_tuple(classname, fieldnames):
    # Наполняем словарь аксессоров свойств полей
    cls_dict = { name: property(operator.itemgetter(n))
                  for n, name in enumerate(fieldnames) }

    # Создаём функцию __new__ и добавляем её в словарь класса
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Создаём класс
    cls = types.new_class(classname, (tuple,), {}, 
                          lambda ns: ns.update(cls_dict))

    # Устанавливаем модуль класса на модуль вызывающего
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls
```

В последней части этого кода используется так называемый «фреймхак», который применяет `sys._getframe()` для получения имени модуля вызывавшего

(caller). Ещё один пример фреймхака показан в [рецепте 2.15](#).

Следующий пример демонстрирует работу этого кода:

```
>>> Point = named_tuple('Point', ['x', 'y'])
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print('%s %s' % p)
4 5
>>>
```

Важный аспект использованного в этом приёме рецепта — правильная поддержка метаклассов. Вы можете подумывать о создании класса напрямую путём прямого создания экземпляра метакласса. Например:

```
Stock = type('Stock', (), cls_dict)
```

Проблема такого подхода заключается в том, что он пропускает некоторые критически важные шаги, такие как вызов метода метакласса `__prepare__()`. А вот `types.new_class()` позволяет удостовериться, что все необходимые шаги инициализации будут произведены. Например, функция обратного вызова, которая передается как четвёртый аргумент в `types.new_class()`, принимает объект отображения, который возвращается методом `__prepare__()`.

Если вы всего лишь хотите выполнить подготовительный шаг, используйте `types.prepare_class()`. Например:

```
import types

metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metaclass': t
```

Этот код обнаруживает подходящий метакласс и вызывает его метод `__prepare__()`. В результате возвращаются метакласс, оставшиеся именованные

аргументы и подготовленное пространство имён.

За дополнительной информацией обратитесь к [PEP 3115](#), а также [документации Python](#).

9.19. Инициализация членов класса во время определения

Задача

Вы хотите инициализировать части определения класса один раз, во время определения класса, а не при создании экземпляров.

Решение

Выполнение действий по инициализации или начальному присваиванию значений во время определения класса — это классический пример использования метаклассов. Метакласс «срабатывает» как раз в момент определения, когда вы можете выполнить дополнительные шаги.

Вот пример, который использует эту идею для создания классов, похожих на именованные кортежи из модуля *collections*:

```
import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(cls._fields)))
        return super().__new__(cls, args)
```

Этот код позволяет определять простые структуры на основе кортежей:

```
class Stock(StructTuple):
```

```
_fields = ['name', 'shares', 'price']

class Point(StructTuple):
    _fields = ['x', 'y']
```

Вот как они работают:

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
('ACME', 50, 91.1)
>>> s[0]
'ACME'
>>> s.name
'ACME'
>>> s.shares * s.price
4555.0
>>> s.shares = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Обсуждение

В этом рецепте класс *StructTupleMeta* принимает список имён атрибутов в атрибуте класса *_fields* и превращает их в методы свойства (*property*), которые осуществляют доступ к конкретному слоту кортежа. Функция *operator.itemgetter()* создает функцию доступа (акцессор), а функция *property()* превращает ее в свойство.

Главная сложность этого рецепта — знать, когда происходят различные шаги инициализации. Метод *__init__()* в *StructTupleMeta* вызывается только один раз для каждого определяемого класса. Аргумент *cls* — это класс, который только что был определён. Код использует переменную класса *_fields*, чтобы взять новый определённый класс и добавить в него несколько новых частей.

Класс *StructTuple* служит общим базовым классом для пользователей, которые его наследуют. Метод *__new__()* в этом классе отвечает за создание новых экземпляров. Использование *__new__()* здесь несколько необычное, поскольку частично связано с тем фактором, что мы модифицируем сигнатуру вызова кортежей, чтобы получить возможность создавать экземпляры с помощью обычно выглядящих вызовов:

```
s = Stock( 'ACME' , 50 , 91.1)           # OK
s = Stock( ('ACME' , 50 , 91.1))        # Ошибка
```

В отличие от `__init__()`, метод `__new__()` вызывается до того, как экземпляр создан. Поскольку кортежи неизменяемы, невозможно внести какие-либо изменения в них после создания. Функция `__init__()` запускается в процессе инициализации слишком поздно, чтобы выполнить эту задачу. Именно поэтому был определён метод `__new__()`.

Хотя это короткий рецепт, его внимательное изучение вознаградит читателя глубоким пониманием того, как определяются классы в Python, как создаются экземпляры, а также знанием точек, в которых вызываются различные методы метаклассов и классов.

[PEP 422](#) может предоставить дополнительные средства для решения поставленной в этом рецепте задачи. Однако на момент написания книги он не был внедрён. (Прим. пер.: на момент перевода предложение отозвано).

9.20. Реализация множественной диспетчеризации с помощью аннотаций функций

Задача

Вы узнали об аннотациях аргументов функций и задумались, нельзя ли использовать их для реализации множественной диспетчеризации (перегрузки методов) на основе типов. Однако вы не уверены, что тут нужно использовать (и хорошая ли это идея в принципе).

Решение

Этот рецепт базируется на простом наблюдении — поскольку Python позволяет аннотировать аргументы, то можно написать такой код:

```
class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)
    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)
```

```
s = Spam()
s.bar(2, 3)           # Выводим Bar 1: 2 3
s.bar('hello')        # Выводим Bar 2: hello 0
```

Вот начало решения, которое делает то же самое, но с использованием комбинации метаклассов и дескрипторов:

```
# multiple.py
import inspect
import types

class MultiMethod:
    """
    Представляет один мульти метод.
    """

    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        """
        Регистрирует новый метод как мульти метод
        """

        sig = inspect.signature(meth)

        # Создание сигнатуры типа из аннотаций методов
        types = []
        for name, parm in sig.parameters.items():
            if name == 'self':
                continue
            if parm.annotation is inspect.Parameter.empty:
                raise TypeError(
                    'Argument {} must be annotated with a type'.format(r
                )
            if not isinstance(parm.annotation, type):
                raise TypeError(
                    'Argument {} annotation must be a type'.format(r
                )
            if parm.default is not inspect.Parameter.empty:
                self._methods[tuple(types)] = meth
                types.append(parm.annotation)

        self._methods[tuple(types)] = meth

    def __call__(self, *args):
        """
        Вызов метода базируется на сигнатуре типа аргументов
        """

        types = tuple(type(arg) for arg in args[1:])
```

```

meth = self._methods.get(types, None)
if meth:
    return meth(*args)
else:
    raise TypeError('No matching method for types {}'.format(types))

def __get__(self, instance, cls):
    """
    Метод дескриптора, необходимый для работы вызовов в классе
    ...
    if instance is not None:
        return types.MethodType(self, instance)
    else:
        return self

class MultiDict(dict):
    """
    Специальный словарь для создания мульти методов в метаклассе
    ...
    def __setitem__(self, key, value):
        if key in self:
            # Если ключ уже существует, он должен быть мульти методом
            # или вызываемым объектом
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)

class MultipleMeta(type):
    """
    Метакласс, который позволяет множественную диспетчеризацию методов
    ...
    def __new__(cls, clsname, bases, clsdic):
        return type.__new__(cls, clsname, bases, dict(cldic))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()

```

Чтобы использовать этот класс, напишите такой код:

```

class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):

```

```

        print('Bar 1:', x, y)
def bar(self, s:str, n:int = 0):
    print('Bar 2:', s, n)

# Пример: перегруженный __init__
import time
class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day
    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)

```

Вот интерактивный сеанс, в котором мы проверяем, что всё работает:

```

>>> s = Spam()
>>> s.bar(2, 3)
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "multiple.py", line 42, in __call__
      raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 'str'>)
>>> # Перегруженный __init__
>>> d = Date(2012, 12, 21)
>>> # Получить сегодняшнюю дату
>>> e = Date()
>>> e.year
2012
>>> e.month
12
>>> e.day
3
>>>

```

Обсуждение

Если честно, то в этом рецепте слишком много магии, чтобы применять его в реальном мире. Однако он позволяет погрузиться во внутреннюю работу метаклассов и дескрипторов, улучшая ваше понимание этих концепций. Так

что если вы и не будете применять этот рецепт напрямую, многие его идеи могут повлиять на другие приёмы программирования, использующие метаклассы, дескрипторы и аннотации функций.

Основная идея этой реализации относительно проста. Метакласс *MultipleMeta* использует свой метод `__prepare__()`, чтобы предоставить кастомный словарь класса в качестве экземпляра *MultiDict*. В отличие от обычного словаря, *MultiDict* во время присваивания значений проверяет, существуют ли уже эти записи. Если они уже существуют, дублированные записи сливаются вместе внутри экземпляра *MultiMethod*.

Экземпляры *MultiMethod* собирают методы путём построения отображения из сигнатур типов в функции. Во время создания аннотации функции используются для сбора этих сигнатур и построения отображения. Это происходит в методе *MultiMethod.register()*. Важнейший аспект этого рецепта заключается в том, что для мультиметодов типы должны быть определены на всех аргументах, иначе возникнет ошибка.

Чтобы заставить экземпляры *MultiMethod* эмулировать поведение вызываемого объекта, в них реализован метод `__call__()`. Этот метод строит кортеж типов из всех аргументов, за исключением *self*, ищет метод во внутреннем отображении и вызывает подходящий метод. Метод `__get__()` нужен, чтобы заставить экземпляры *MultiMethod* правильно работать внутри определений классов. В данной реализации он был использован для создания правильных связанных методов. Например:

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

Будьте уверены, в этом рецепте много «движущихся частей». Однако это не спасает от большого количества ограничений. Например, это решение не работает с именованными аргументами:

```
>>> s.bar(x=2, y=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'
>>> s.bar(s='hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>
```

Способ добавить поддержку, возможно, нашёлся бы, но он потребовал бы абсолютно другого подхода к отображению методов. Проблема в том, что именованные аргументы не поставляются в каком-то конкретном порядке. При смещивании с позиционными аргументами вы получите беспорядочный клубок, который вы как-то должны будете распутывать в методе `__call__()`.

Этот рецепт также серьёзно ограничен в том, что касается поддержки наследования. Например, что-то такое работать не будет:

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class Spam(metaclass=MultipleMeta):
    def foo(self, x:A):
        print('Foo 1:', x)
    def foo(self, x:C):
        print('Foo 2:', x)
```

Причина, по которой всё ломается, такова: аннотация `x:A` не совпадает с экземплярами, которые являются подклассами (такими, как экземпляры `B`). Например:

```
>>> s = Spam()
>>> a = A()
>>> s.foo(a)
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
```

```
>>> s.foo(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "multiple.py", line 44, in __call__
      raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class '__main__.B'>,)
>>>
```

В качестве альтернативы использованию метаклассов и аннотаций можно реализовать похожий рецепт, использующий декораторы. Например:

```
import types

class multimethod:
    def __init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__)
            if func.__defaults__:
                for n in range(ndefaults+1):
                    self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)

    def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self
```

Чтобы использовать версию на базе декораторов, напишите такой код:

```
class Spam:
    @multimethod
    def bar(self, *args):
        Если нет совпадений, вызывается дефолтный метод
        raise TypeError('No matching method for bar')
```

```
@bar.match(int, int)
def bar(self, x, y):
    print('Bar 1:', x, y)

@bar.match(str, int)
def bar(self, s, n = 0):
    print('Bar 2:', s, n)
```

Решение на базе декораторов страдает от тех же ограничений, что и предыдущая реализация (от отсутствия поддержки именованных аргументов и поломанного наследования).

При прочих равных, вероятно, лучше держаться подальше от множественной диспетчеризации в коде общего назначения. Существуют особые ситуации, где это может иметь смысл — например, в программах, где диспетчеризация методов базируется на каком-то сопоставлении с образцом (pattern matching). Например, описанный в [рецепте 8.21](#). паттерн (шаблон) проектирования «Посетитель» может быть преобразован в класс, который неким образом использует множественную диспетчеризацию. Однако никогда не будет дурной идеей остановиться на более простом подходе (просто использовать методы с разными именами).

Идеи, касающиеся различных способов реализации множественной диспетчеризации, бродят в сообществе Python уже много лет. Достойной стартовой точкой для вхождения в курс этой дискуссии станет пост Гвидо ван Россума [«Мультиметоды в Python за пять минут»](#).

9.21. Избежание повторяющихся методов свойств

Задача

Вы пишете классы, где постоянно вынуждены повторять определение методов свойств, которые выполняют обычные задачи, такие как проверка типов. Вы хотели бы упростить код и убрать из него повторения.

Решение

Рассмотрите простой класс, в котором атрибуты обёрнуты методами-

свойствами:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('name must be a string')
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not isinstance(value, int):
            raise TypeError('age must be an int')
        self._age = value
```

Как вы можете видеть, куча кода написана просто для того, чтобы внедрить принудительную проверку типов значений атрибутов. Всякий раз, когда вы видите такой код, вы должны поискать пути упрощения. Возможный подход — создать функцию, которая определяет свойство и возвращает его.

Например:

```
def typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def prop(self):
        return getattr(self, storage_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
            raise TypeError('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)
    return prop

# Пример
```

```
class Person:  
    name = typed_property('name', str)  
    age = typed_property('age', int)  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Обсуждение

Этот рецепт иллюстрирует важную возможность вложенных функций (замыканий) — с точки зрения работы они во многом похожи на макросы. Функция `typed_property()` в этом примере может показаться странной, но она просто генерирует для вас код свойства и возвращает получившийся объект свойства. Так что при использовании в классе она работает точно так же, как если бы код, появляющийся внутри `typed_property()`, был помещен в само определение класса. Хотя методы свойства (геттер и сеттер) обращаются к локальным переменным, таким как `name`, `expected_type` и `storage_name`, это нормально — эти значения хранятся в замыкании.

Этот рецепт можно интересным образом «прокачать», используя функцию `functools.partial()`. Например, вы можете сделать так:

```
from functools import partial  
  
String = partial(typed_property, expected_type=str)  
Integer = partial(typed_property, expected_type=int)  
  
# Пример  
class Person:  
    name = String('name')  
    age = Integer('age')  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Тут код начинает походить на дескриптор системы типов, показанный в [рецепте 8.13](#).

9.22. Лёгкий способ определения менеджеров контекста

Задача

Вы хотите реализовать новые менеджеры контекста для использования с инструкцией *with*.

Решение

Один из самых простых способов написания нового менеджера контекста — использовать декоратор `@contextmanager` из модуля `contextlib`. Вот пример менеджера контекста, подсчитывающего время выполнения блока кода:

```
import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print(' {}: {}'.format(label, end - start))

# Пример
with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1
```

В функции `timethis()` весь код перед `yield` выполняется как метод `__enter__()` менеджера контекста. Весь код после `yield` выполняется как метод `__exit__()`. Если имеет место исключение, оно возбуждается в инструкции `yield`.

Вот немного более продвинутый пример менеджера контекста, который реализует некую транзакцию на объекте списка:

```
@contextmanager
def list_transaction(orig_list):
    working = list(orig_list)
    yield working
    orig_list[:] = working
```

Идея в том, что изменения вносятся в список только в том случае, если при выполнении всего блока кода не возбуждаются исключения. Вот пример:

```
>>> items = [1, 2, 3]
>>> with list_transaction(items) as working:
...     working.append(4)
...     working.append(5)
...
>>> items
[1, 2, 3, 4, 5]
>>> with list_transaction(items) as working:
...     working.append(6)
...     working.append(7)
...     raise RuntimeError('oops')
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: oops
>>> items
[1, 2, 3, 4, 5]
>>>
```

Обсуждение

Чтобы написать менеджер контекста, обычно вы определяете класс с методами `__enter__()` и `__exit__()`:

```
import time

class timethis:
    def __init__(self, label):
        self.label = label
    def __enter__(self):
        self.start = time.time()
    def __exit__(self, exc_ty, exc_val, exc_tb):
        end = time.time()
        print('{}: {}'.format(self.label, end - self.start))
```

Хотя это несложно, но все же более утомительно, чем писать простые функции с декоратором `@contextmanager`.

`@contextmanager` на самом деле используется для написания замкнутых на себя функций с управлением контекстом. Если вам нужно реализовать поддержку инструкции `with` каким-либо объектом (например, файлом, сетевым соединением или блокировкой), вам всё равно придется отдельно реализовать методы `__enter__()` и `__exit__()`.

9.23. Выполнение кода с локальными

побочными эффектами

Задача

Вы используете `exec()` для выполнения фрагмента кода в области видимости вызывающего (caller), но после выполнения никаких результатов не видно.

Решение

Чтобы лучше понять проблему, проведите небольшой эксперимент. Для начала выполним фрагмент кода в глобальном пространстве имён:

```
>>> a = 13
>>> exec('b = a + 1')
>>> print(b)
14
>>>
```

А теперь попробуем сделать то же самое внутри функции:

```
>>> def test():
...     a = 13
...     exec('b = a + 1')
...     print(b)
...
>>> test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in test
      NameError: global name 'b' is not defined
>>>
```

Как вы можете видеть, программа падает с ошибкой `NameError()` — практически так же, как если бы инструкция `exec()` вообще не была выполнена. Это может стать проблемой, если вы захотите использовать результат `exec()` в последующих вычислениях.

Чтобы исправить проблемы такого типа, вам нужна функция `locals()`. Она позволяет получить словарь локальных переменных перед вызовом `exec()`. Сразу после этого вы можете извлечь изменённые значения из данного словаря. Например:

```
>>> def test():
...     a = 13
...     loc = locals()
...     exec('b = a + 1')
...     b = loc['b']
...     print(b)
...
>>> test()
14
>>>
```

Обсуждение

Правильное использование `exec()` на практике — довольно сложная штука. На самом деле, в большинстве случаев, когда вы могли бы задуматься о применении `exec()`, можно поискать более элегантное решение (например, декораторы, замыкания, метаклассы и т.д.)

Однако если вы должны использовать именно `exec()`, этот рецепт подчёркивает некоторые тонкие аспекты того, как делать это правильно. По умолчанию `exec()` выполняет код в локальной и глобальной области видимости вызывающего. Однако внутри функций локальная область видимости передаётся в `exec()` как словарь, который является копией настоящих локальных переменных. Так что если код, который выполняется в `exec()`, вносит какие-либо изменения, эти изменения никогда не отражаются на настоящих локальных переменных. Вот ещё один пример, который демонстрирует этот эффект:

```
>>> def test1():
...     x = 0
...     exec('x += 1')
...     print(x)
...
>>> test1()
0
>>>
```

Когда вы вызываете `locals()`, чтобы получить локальные переменные, как показано в решении, вы получаете копию, которая передана в `exec()`. Путём инспектирования значений словаря после выполнения, вы можете получить изменённые значения. Вот эксперимент, который это демонстрирует:

```
>>> def test2():
...     a = 13
...     loc = locals()
...     exec('a += 1')
...     print(loc['a'])
...
>>> test2()
14
>>>
```

```
...     x = 0
...     loc = locals()
...     print('before:', loc)
...     exec('x += 1')
...     print('after:', loc)
...     print('x =', x)
...
>>> test2()
before: {'x': 0}
after: {'loc': {...}, 'x': 1}
x = 0
>>>
```

Обратите внимание на вывод на последнем шаге. До тех пор, пока вы не скопировали значение из *loc* обратно в *x*, переменная оставалась неизменной.

При использовании *locals()* вам нужно следить за порядком выполнения операций. Каждый раз, когда *locals()* вызывается, она берёт текущие значения локальных переменных и переписывает соответствующие записи в словаре. Понаблюдайте за этим экспериментом:

```
>>> def test3():
...     x = 0
...     loc = locals()
...     print(loc)
...     exec('x += 1')
...     print(loc)
...     locals()
...     print(loc)
...
>>> test3()
{'x': 0}
{'loc': {...}, 'x': 1}
{'loc': {...}, 'x': 0}
>>>
```

Обратите внимание, как последний вызов *locals()* вызывает перезаписывание *x*.

В качестве альтернативы использованию *locals()*, вы можете создать собственный словарь и передать его в *exec()*. Например:

```
>>> def test4():
...     a = 13
...     loc = { 'a' : a }
...     alb = { }
```

```
...     exec('b = a + 1', glb, loc)
...     b = loc['b']
...     print(b)
...
>>> test4()
14
>>>
```

Для большинства случаев использования `exec()` это, вероятно, будет хорошей практикой. Вам просто нужно убедиться, что глобальный и локальный словари правильно инициализированы именами, к которым будет обращаться выполняемый код.

И последнее: перед использованием `exec()` стоит подумать об альтернативах. Многие подобных задачи успешно решаются с помощью замыканий, декораторов, метаклассов или других приёмов метaprogramмирования.

9.24. Парсинг и анализ исходного кода Python

Задача

Вы хотите писать программы, которые парсят и анализируют исходный код Python.

Решение

Многие программисты знают, что Python может выполнять или вычислять исходный код, предоставленный в форме строки. Например:

```
>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
```

>>>

Однако можно использовать модуль `ast`, чтобы скомпилировать исходный код Python в абстрактное синтаксическое дерево (AST), которое может быть проанализировано. Например:

```
>>> import ast
>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
right=Name(id='x', ctx=Load())))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None)), orelse=[])])"
>>>
```

Анализ исходного дерева требует от вас некоторых усилий. Он состоит из коллекции AST-узлов. Самый простой способ работы с этим узлами — определить класс-посетитель, который реализует различные методы `visit_NodeName`, где `NodeName` совпадает с интересующим узлом. Вот пример такого класса, который записывает информацию о том, какие имена были загружены, сохранены и удалены.

```
import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()
        self.deleted = set()
    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
```

```
    elif isinstance(node.ctx, ast.Del):
        self.deleted.add(node.id)

# Пример использования
if __name__ == '__main__':
    # Какой-то код
    code = '''
for i in range(10):
    print(i)
del i
'''

    # Парсим в AST
    top = ast.parse(code, mode='exec')

    # Скармливаем AST анализатору использования имён
    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)
```

Если вы запустите эту программу, то получите такой вывод:

```
Loaded: {'i', 'range', 'print'}
Stored: {'i'}
Deleted: {'i'}
```

Наконец, AST могут быть скомпилированы и выполнены с использованием функции `compile()`. Например:

```
>>> exec(compile(top, '<stdin>', 'exec'))
0
1
2
3
4
5
6
7
8
9
>>>
```

Обсуждение

Тот факт, что вы можете анализировать исходный код и извлекать из него информацию, может стать основой для написания различных инструментов

для анализа, оптимизации и верификации кода. Например, вместо передачи фрагментов кода в функцию `exec()`, вы можете превратить его в AST и изучить его, чтобы понять, что он делает. Вы можете также написать инструменты, которые просматривают весь исходный код модуля и производят некий статический анализ.

Стоит отметить, что можно переписать AST, чтобы представить новый код, если вы на самом деле понимаете, что вы делаете. Вот пример декоратора, который «опускает» имена с глобальным доступом в тело функции путём репарсинга исходного кода тела функции, переписывания AST и воссоздания объекта кода функции:

```
# namelower.py
import ast
import inspect

# Узел-посетитель, который «понижает» глобально доступные
# имена в тело функции, делая их локальными переменными.

class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Компилируем некие присвоения для «понижения» констант
        code = '__globals = globals()\n'
        code += '\n'.join("{0} = __globals['{0}']".format(name)
                         for name in self.lowered_names)

    code_ast = ast.parse(code, mode='exec')

    # Инъецируем новые инструкции в тело функции
    node.body[:0] = code_ast.body

    # Сохраняем объект функции
    self.func = node

# Декоратор, который превращает глобальные имена в локальные
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Пропускаем линии исходного кода перед декоратором @lower_names
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Хак, чтобы разобраться с отступами кода
```

```

if src.startswith(' ', '\t')):
    src = 'if 1:\n' + src
top = ast.parse(src, mode='exec')

# Трансформируем AST
cl = NameLower(namelist)
cl.visit(top)

# Выполняем модифицированное AST
temp = {}
exec(compile(top, '', 'exec'), temp, temp)

# Достаём модифицированный объект кода
func.__code__ = temp[func.__name__].__code__
return func
return lower

```

Чтобы использовать этот код, вы могли бы написать что-то такое:

```

INCR = 1

@lower_names('INCR')
def countdown(n):
    while n > 0:
        n -= INCR

```

Декоратор переписывает исходный функции `countdown()`, чтобы она выглядела так:

```

def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
    while n > 0:
        n -= INCR

```

В teste производительности выяснилось, что это заставило функцию работать на 20% быстрее.

Должны ли вы применить этот декоратор ко всем своим функциям? Скорее всего, нет. Однако это хороший пример весьма продвинутых штук, которые доступны через манипуляции AST, исходным кодом и прочие подобные приёмы.

Этот рецепт был вдохновлён похожим рецептом от [ActiveState](#), который работает, опираясь на манипуляции исходным кодом Python. Работа с AST — это высокоуровневый подход, который мог бы быть немного прямолинейнее.

См. следующий рецепт, в котором приведены сведения о байт-коде.

9.25. Дизассемблирование байт-кода Python

Задача

Вы хотите узнать в подробностях, как работает ваш код «под капотом», путём дизассемблирования его в низкоуровневый байт-код с помощью интерпретатора.

Решение

Модуль *dis* можно использовать для вывода результата дизассемблирования любой функции Python. Например:

```
>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...     print('Blastoff!')

>>> import dis
>>> dis.dis(countdown)
 2      0 SETUP_LOOP          39 (to 42)
   >>  3 LOAD_FAST             0 (n)
       6 LOAD_CONST            1 (0)
       9 COMPARE_OP            4 (>)
      12 POP_JUMP_IF_FALSE     41

 3      15 LOAD_GLOBAL           0 (print)
      18 LOAD_CONST            2 ('T-minus')
      21 LOAD_FAST             0 (n)
      24 CALL_FUNCTION          2 (2 positional, 0 keyword pair)
      27 POP_TOP

 4      28 LOAD_FAST             0 (n)
      31 LOAD_CONST            3 (1)
      34 INPLACE_SUBTRACT
      35 STORE_FAST             0 (n)
      38 JUMP_ABSOLUTE          3
   >> 41 POP_BLOCK

 5  >> LOAD_GLOBAL             0 (print)
```

```
LOAD_CONST           4 ('Blastoff!')
CALL_FUNCTION        1 (1 positional, 0 keyword pair)
POP_TOP
LOAD_CONST          0 (None)
RETURN_VALUE

>>>
```

Обсуждение

Модуль *dis* может быть полезен, если вам когда-либо потребуется изучить, что происходит в ваших программах на очень низком уровне (например, если вы пытаетесь разобраться в характеристиках производительности).

Сырой байт-код, интерпретируемый функцией *dis()*, доступен в функциях:

```
>>> countdown.__code__.co_code
b"\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00\x00\x83
\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00g\x03\x00w\x00\x00d\x04\x00\x83
\x01\x00\x01d\x00\x00S"
>>>
```

Если вы захотите интерпретировать этот код самостоятельно, вам понадобятся некоторые из констант, определённых в модуле *opcode*. Например:

```
>>> c = countdown.__code__.co_code
>>> import opcode
>>> opcode.opname[c[0]]
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[3]]
'LOAD_FAST'
>>>
```

Забавно, но в модуле *dis* нет функции, которая облегчила бы вам программную обработку байт-кода. Однако эта функция-генератор примет сырую последовательность байт-кода и превратит её в опкоды (коды операций) и аргументы:

```
import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
```

```

n = len(codebytes)
while i < n:
    op = codebytes[i]
    i += 1
    if op >= opcode.HAVE_ARGUMENT:
        oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
        extended_arg = 0
        i += 2
        if op == opcode.EXTENDED_ARG:
            extended_arg = oparg * 65536
            continue
    else:
        oparg = None
yield (op, oparg)

```

Чтобы использовать эту функцию, напишите такой код:

```

>>> for op, oparg in generate_opcodes(countdown.__code__.co_code):
...     print(op, opcode.opname[op], oparg)
...
120 SETUP_LOOP 39
124 LOAD_FAST 0
100 LOAD_CONST 1
107 COMPARE_OP 4
114 POP_JUMP_IF_FALSE 41
116 LOAD_GLOBAL 0
100 LOAD_CONST 2
124 LOAD_FAST 0
131 CALL_FUNCTION 2
1 POP_TOP None
124 LOAD_FAST 0
100 LOAD_CONST 3
56 INPLACE_SUBTRACT None
125 STORE_FAST 0
113 JUMP_ABSOLUTE 3
87 POP_BLOCK None
116 LOAD_GLOBAL 0
100 LOAD_CONST 4
131 CALL_FUNCTION 1
1 POP_TOP None
100 LOAD_CONST 0
83 RETURN_VALUE None
>>>

```

Это малоизвестный факт, но вы можете заменить сырой байт-код любой функции, какой пожелаете. Это требует некоторых усилий, но вот пример того, как это работает:

```
>>> def add(x, y):
...     return x + y
...
>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Создаем полностью новый объект кода с фейковым байт-кодом
>>> import types
>>> newbytecode = b'xxxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
...     c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_consts,
...     c.co_names, c.co_varnames, c.co_filename, c.co_name,
...     c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault
```

Обрушение интерпретатора — это самый вероятный исход таких безумных экспериментов. Однако разработчики, выполняющие продвинутую оптимизацию и создание инструментов метапрограммирования, могут заниматься этим в реальной жизни. Эта последняя часть иллюстрирует то, как это можно сделать. Вот [ещё один пример кода на ActiveState](#), где показан этот приём в действии.

10. Модули и пакеты

Модули и пакеты — ядро любого крупного проекта (и самого Python). Эта глава фокусируется на распространённых приёмах программирования, использующих модули и пакеты: упорядочивание пакетов, разделение крупных модулей на несколько файлов, создание пространств имён пакетов. Также здесь даны рецепты, которые позволяют кастомизировать работу самой инструкции *import*.

10.1. Создание иерархического пакета модулей

Задача

Вы хотите упорядочить ваш код, поместив его в пакет, состоящий из иерархической коллекции модулей.

Решение

Создать структуру пакета очень просто. Просто организуйте ваш код в структуре каталогов и убедитесь, что каждый каталог содержит файл `__init__.py`. Например:

```
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

Сразу после этого вы сможете выполнять различные инструкции `import`, такие как:

```
import graphics.primitive.line
from graphics.primitive import line
import graphics.formats.jpg as jpg
```

Обсуждение

Определение иерархии модулей не сложнее создания структуры каталогов в файловой системе. В файлы `__init__.py` можно поместить необязательный код инициализации, который запускается, когда происходит обращение к различным уровням пакета.

Например, если у вас есть инструкция `import graphics`, будет импортирован файл `graphics/__init__.py` и сформировано содержание пространства имён `graphics`. Для инструкции `import graphics.formats.jpg` будут импортированы файлы `graphics/__init__.py` и `graphics/formats/__init__.py`, причем перед финальным импортированием файла `graphics/formats/jpg.py`.

Часто можно просто оставить файл `__init__.py` пустым. Однако есть некоторые ситуации, при которых эти файлы могут содержать код. Например, файл `__init__.py` может быть использован для автоматической загрузки подмодулей:

```
# graphics/formats/__init__.py
from . import jpg
from . import png
```

При наличии такого файла пользователь должен будет выполнить одну инструкцию `import graphics.formats` вместо отдельного импортирования `graphics.formats.jpg` и `graphics.formats.png`.

Другой типичный пример использования `__init__.py` — объединение определений из различных файлов в единое логическое пространство имён, что иногда делается при разделении модулей. Это обсуждается в **рецепте 10.4.**

Проницательные программисты заметят, что Python 3.3 всё ещё выполняет импортирование пакетов, даже если файлы `__init__.py` отсутствуют. Если вы не определяете `__init__.py`, то вы создаёте так называемый «пакет пространства имён», как описано в **рецепте 10.5.** При прочих равных, включайте файлы `__init__.py`, если вы просто начинаете создание нового пакета.

10.2. Контроль импортирования

Задача

Вам нужен полный контроль над именами, которые экспортятся из модуля или пакета, когда пользователь применяет инструкцию `from module import *`.

Решение

Определите переменную `__all__` в вашем модуле, где явно перечислите экспортируемые имена. Например:

```
# somemodule.py
def spam():
```

```
pass

def grok():
    pass

blah = 42

# Экспортирует только 'spam' и 'grok'
__all__ = ['spam', 'grok']
```

Обсуждение

Хотя использование `from module import *` не поощряется, это всё еще часто встречающийся приём при работе с модулями, в которых определено очень много имён. Если вы ничего не предпримете, эта форма будет экспортировать все имена, которые не начинаются с нижнего подчёркивания. С другой стороны, если вы определите `__all__`, то только явно перечисленные в ней имена будут экспортированы.

Если вы определите `__all__` как пустой список, ничего экспортироваться не будет. Если `__all__` содержит неопределённые имена, то будет возбуждено исключение `AttributeError`.

10.3. Импортирование подмодулей пакета с использованием относительных имён

Задача

У вас есть упорядоченный в пакет код, и вы хотите импортировать подмодуль из одного из подмодулей другого пакета без жесткого прописывания («хардкодинга») имени пакета в инструкции импортирования.

Решение

Чтобы импортировать модули пакета из других модулей в том же пакете, используйте импортирование относительно пакета. Предположим, например, что у вас есть пакет `mypackage`, который организован следующим образом:

```
mypackage/
    __init__.py
```

A/

```
__init__.py  
spam.py  
grok.py
```

B/

```
__init__.py  
bar.py
```

Если модуль *mypackage.A.spam* пожелает импортировать модуль *grok*, размещённый в том же каталоге, он должен будет обзавестись такой инструкцией:

```
# mypackage/A/spam.py  
  
from . import grok
```

Если тот же модуль захочет импортировать модуль *B.bar*, размещённый в другом каталоге, мы можем использовать такую инструкцию:

```
# mypackage/A/spam.py  
  
from ..B import bar
```

Обе показанных инструкции импортирования работают относительно местоположения файла *spam.py* и не включают имя пакета высшего уровня.

Обсуждение

Внутри пакетов импортирование, задействующее модули того же пакета, может использовать либо полные абсолютные имена, либо относительное импортирование через показанный выше синтаксис. Например:

```
# mypackage/A/spam.py  
from mypackage.A import grok      # OK  
from . import grok                # OK  
import grok                      # Ошибка (не найден)
```

Недостаток использования абсолютного имени, такого как *mypackage.A*, заключается в том, что оно жёстко прописывает имя пакета высшего уровня в исходный код. Это, в свою очередь, делает ваш код более хрупким и затрудняет работу по реорганизации. Например, если вы когда-либо захотите поменять имя пакета, вам придется пройтись по всем файлам и поправить исходный код. Похожим образом жёстко прописанные имена затрудняют

перемещение кода. Например, если кто-то захочет установить две разных версии пакета под различными именами, то при использовании относительного импортирования всё будет работать хорошо, но абсолютные имена не позволят этого сделать.

Синтаксис инструкции *import .* и .. может показаться забавным, но предлагаем вам думать о нём как об определении имени каталога. . значит «ищи в текущем каталоге», а .. — «ищи в каталоге /B». Этот синтаксис работает только с формой *from*. Например:

```
from . import grok      # OK
import .grok            # Ошибка
```

Хотя это выглядит так, будто вы можете перемещаться по файловой системе через относительное импортирование, но на самом деле это не позволит вам выйти из каталога, в котором определён пакет. Это означает, что комбинирование имён с точками, которое заставляет производить импортирование из-за пределов пакета, вызовет ошибку.

Также стоит заметить, что относительное импортирование работает только для модулей, которые размещены внутри подходящего пакета. В частности, оно не работает внутри простых модулей, размещенных на верхнем уровне скриптов. Оно также не работает, если части пакета исполняются напрямую, как скрипты. Например:

```
% python3 mypackage/A/spam.py
# Относительное импортирование не работает
```

С другой стороны, если вы выполните предыдущий скрипт, передав в Python опцию *-m*, относительное импортирование будет работать правильно. Например:

```
% python3 -m mypackage.A.spam
# Относительное импортирование работает
```

За дополнительными сведениями об относительном импортировании пакетов обратитесь к [PEP 328](#).

10.4. Разделение модуля на несколько файлов

Задача

У вас есть модуль, который вы хотели бы разделить на несколько файлов. Однако вы хотели бы сделать это, не ломая существующий код — сохранив отдельные файлы объединёнными в один логический модуль.

Решение

Программный модуль можно разделить на отдельные файлы путём превращения в пакет. Рассмотрим следующий простой модуль:

```
# mymodule.py

class A:
    def spam(self):
        print('A.spam')

class B(A):
    def bar(self):
        print('B.bar')
```

Предположим, что вы хотите разделить *mymodule.py* на два файла, по одному на каждое определение класса. Чтобы это сделать, начните с замены файла *mymodule.py* на каталог с именем *mymodule*. В этом каталоге создайте следующие файлы:

```
mymodule/
__init__.py
a.py
b.py
```

В файл *a.py* поместите этот код:

```
# a.py

class A:
    def spam(self):
        print('A.spam')
```

В файл *b.py* поместите этот код:

```
# b.py
```

```
from .a import A

class B(A):
    def bar(self):
        print('B.bar')
```

И, наконец, склейте в файле `__init__.py` оба эти файла:

```
# __init__.py

from .a import A
from .b import B
```

Если вы выполните эти шаги, получившийся пакет `mymodule` будет работать, как единый логический модуль:

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>> b = mymodule.B()
>>> b.bar()
B.bar
>>>
```

Обсуждение

Основная проблема в этом рецепте — понять, хотите ли вы, чтобы пользователи работали с большим количеством маленьких модулей или одним модулем. Например, если у вас обширная кодовая база, вы можете просто разбить всё на отдельные файлы и заставить пользователей писать множество инструкций `import`:

```
from mymodule.a import A
from mymodule.b import B
...
```

Это работает, но на пользователей возлагается бремя — они должны знать, где расположены различные части. Часто проще всё объединить и разрешить единый импорт:

```
from mymodule import A, B
```

В последнем случае о *mymodule* обычно думают, как о большом файле с исходным кодом. Однако этот рецепт показывает, как сшить в единое логическое пространство имён множество файлов. Ключ к этому — создать каталог пакета и использовать для склеивания частей *init.py*.

Когда модуль разделяется, вы должны внимательно отнестись перекрёстным ссылкам имён файлов. В этом рецепте, например, класс *B* нуждается в доступе к классу *A* как к базовому классу. Чтобы добиться этого, используется относительное импортирование на уровне пакета *from . import A*.

Относительное импортирование на уровне пакета использовано в этом рецепте для устранения жёсткого прописывания («хардкодинга») имени модуля высшего уровня в исходном коде. Это делает проще переименование и перемещение модуля (см. [рецепт 10.3.](#))

Для этого рецепта есть интересное дополнение, которое вводит концепцию «ленивого» импортирования. Как показано выше, *__init__.py* импортирует все требуемые подкомпоненты за один раз. Однако для очень большого модуля может быть целесообразно импортировать компоненты по мере необходимости. Чтобы сделать это, немного изменим *__init__.py*:

```
# __init__.py

def A():
    from .a import A
    return A()

def B():
    from .b import B
    return B()
```

В этой версии классы *A* и *B* заменены функциями, которые загружают нужные классы при первом доступе к ним. С точки зрения пользователя это не слишком большое отличие. Например:

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

Главный недостаток ленивой загрузки в том, что могут поломаться

наследования и проверка типов. Например, вы могли бы немного изменить код:

```
if isinstance(x, mymodule.A):      # Ошибка
...
if isinstance(x, mymodule.a.A):    # Ok
...
```

Если вам нужен пример ленивой загрузки из реального мира, посмотрите на исходный код *multiprocessing/_init__.py* в стандартной библиотеке.

10.5. Заставляем отдельные каталоги с кодом импортироваться под общим пространством имён

Задача

У вас большая кодовая база с частями, которые могут поддерживаться и распространяться разными людьми. Каждая часть организована в виде каталога файлов (как пакет). Однако вместо того, чтобы устанавливать каждую часть отдельно, как пакет с отдельным именем, вы хотели бы объединить все части под общим префиксом пакета.

Решение

По сути, задача сводится к тому, чтобы определить пакет Python высшего уровня, который будет служить пространством имён для большой коллекции отдельно поддерживаемых подпакетов. Эта задача часто возникает в больших фреймворках, когда разработчики хотят поощрить пользователей разрабатывать пакеты плагинов или аддонов.

Чтобы объединить отдельные каталоги под общим пространством имён, вы должны организовать код так же, как и в обычном пакете Python, но опустить файлы *_init__.py* в каталогах, где компоненты будут объединяться.

Предположим, у вас есть два различных каталога с кодом Python:

```
foo-package/
  spam/
    blah.py
```

```
bar-package/
    spam/
        grok.py
```

В этих каталогах имя *spam* используется в качестве общего пространства имен. Обратите внимание, что файл *__init__.py* отсутствует в обоих каталогах.

Теперь посмотрим, что будет, если вы добавите оба пакета (*foo-package* и *bar-package*) к пути поиска модулей Python и попробуете импортировать:

```
>>> import sys
>>> sys.path.extend(['foo-package', 'bar-package'])
>>> import spam.blah
>>> import spam.grok
>>>
```

Волшебным образом два разных каталога пакетов слились вместе, и вы можете импортировать либо *spam.blah*, либо *spam.grok*. Всё работает.

Обсуждение

Механизм, который здесь работает, известен под названием «пакет пространства имен». По сути, пакет пространства имен — это специальный пакет, разработанный для слияния различных каталогов с кодом под общим пространством имен. Для крупных фреймворков это может быть весьма полезно, поскольку позволяет разбить части фреймворка на отдельно устанавливаемые скачиваемые файлы. Это также позволяет людям проще делать сторонние аддоны и другие расширения.

Ключ к созданию пакета пространства имен — отсутствие файлов *__init__.py* в каталоге высшего уровня, который служит общим пространством имен. Отсутствие *__init__.py* вызывает интересный эффект при импортировании пакета. Вместо того, чтобы выкинуть ошибку, интерпретатор начинает создавать список всех каталогов, которые содержат совпадающее имя пакета. Затем создаётся специальный модуль-пакет пространства имен, и в его переменной *__path__* сохраняется доступная только для чтения копия списка каталогов. Например:

```
>>> import spam
>>> spam.__path__
[NamespacePath(['foo-package/spam', 'bar-package/spam'])]
```

```
>>>
```

Каталоги из `__path__` используются при определении следующих подкомпонентов пакета (например, при импортировании `spam.grok` или `spam.blah`).

Важная возможность пакетов пространств имён заключается в том, что кто угодно может расширить пространство имён своим собственным кодом. Например, предположим, что вы создали собственный каталог с кодом:

```
my-package/
    spam/
        custom.py
```

Если вы добавите ваш каталог в `sys.path` вместе с другими пакетами, он бесшовно сольется с другими каталогами пакета `spam`:

```
>>> import spam.custom
>>> import spam.grok
>>> import spam.blah
>>>
```

Для целей отладки важно знать, что основной способ сказать, является ли пакет пакетом пространства имён, — это проверить его атрибут `__file__`. Если он отсутствует, пакет является пространством имён. Это также будет отражено в строке репрезентации (в ней будет слово “namespace”):

```
>>> spam.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
>>> spam
<module 'spam' (namespace)>
>>>
```

Дополнительную информацию о пакетах пространств имён вы можете найти в [PEP 420](#).

10.6. Перезагрузка модулей

Задача

Вы хотите перезагрузить уже загруженный модуль, потому что внесли изменения в его исходный код.

Решение

Чтобы перезагрузить ранее загруженный модуль, используйте `imp.reload()`.

Например:

```
>>> import spam
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

Обсуждение

Перезагрузка модуля часто используется во время отладки и разработки, но в общем случае её небезопасно применять в продакшне, поскольку она не всегда работает так, как вы ожидаете.

«Под капотом» операция `reload()` стирает содержимое словаря модуля и обновляет его путём выполнения заново исходного кода модуля.

Идентификационные данные самого объекта модуля остаются без изменений. Так что эта операция обновляет модуль везде в программе, где он импортирован.

Однако `reload()` не обновляет определения, которые были импортированы с помощью таких инструкций, как `from module import name`. Для примера рассмотрим такой код:

```
# spam.py

def bar():
    print('bar')

def grok():
    print('grok')
```

Теперь начнём интерактивный сеанс:

```
>>> import spam
>>> from spam import grok
```

```
>>> spam.bar()
bar
>>> grok()
grok
>>>
```

Не покидая Python, отредактируйте исходный код `spam.py` так, чтобы функция `grok()` стала такой:

```
def grok():
    print('New grok')
```

Теперь вернитесь к интерактивному сеансу, выполните перезагрузку и попробуйте провести такой эксперимент:

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok()          # Обратите внимание на старый вывод
grok
>>> spam.grok()    # Обратите внимание на новый вывод
New grok
>>>
```

В этом примере вы видите, что загружены две версии `grok()`. Обычно это не то, чего вы хотите, и такие фокусы могут привести к очень большой головной боли.

По этой причине перезагрузки модулей в продакшне стоит избегать. Оставьте этот приём для отладки или интерактивных сеансов, где вы экспериментируете и пробуете разные штуки.

10.7. Создание каталога или zip-архива, запускаемых как единственный скрипт

Задача

Ваша программа превратилась из простого скрипта в приложение с множеством файлов. При этом вы хотели бы предоставить пользователям простой способ запуска программы.

Решение

Если ваше приложение выросло в набор из множества файлов, вы можете поместить его в собственный каталог и добавить в него файл `__main__.py`. Например, вы можете создать такой каталог:

```
myapplication/
    spam.py
    bar.py
    grok.py
    __main__.py
```

Если файл `__main__.py` присутствует, вы просто запускаете интерпретатор Python в каталоге высшего уровня:

```
bash % python3 myapplication
```

Интерпретатор выполнит `__main__.py` как главную программу.

Этот приём также работает, если вы упаковали весь ваш код в zip-архив. Например:

```
bash % ls
spam.py
bar.py
grok.py
__main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

Обсуждение

Создание каталога или zip-архива и добавление файла `__main__.py` — один из возможных путей для упаковки крупного Python-приложения. Он немного отличается от пакета тем, что код не предназначен для использования в качестве стандартного библиотечного модуля, установленного в библиотеку Python. Вместо этого мы получаем просто набор кода, который вы передаете кому-либо, и этот кто-то может запускать эту программу.

Поскольку каталоги и zip-архивы немного отличаются от обычных файлов, вы также можете пожелать добавить вспомогательный скрипт командной

оболочки, чтобы облегчить запуск. Например, если код лежит в файле *myapp.zip*, вы можете написать такой скрипт:

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

10.8. Чтение файлов с данными внутри пакета

Задача

Ваш пакет включает файл с данными, который должен прочесть ваш код. Вам нужно обеспечить максимальную переносимость.

Решение

Предположим, что у вас есть пакет с такой организацией файлов:

```
mypackage/
    __init__.py
    somedata.dat
    spam.py
```

Теперь предположим, что файл *spam.py* хочет прочитать содержимое файла *somedata.dat*. Чтобы это сделать, используйте такой код:

```
# spam.py

import pkgutil
data = pkgutil.get_data(__package__, 'somedata.dat')
```

Получившаяся переменная *data* будет байтовой строкой с «сырым» (raw) содержимым файла.

Обсуждение

Чтобы прочесть файл с данными, вы можете написать код, который использует встроенные функции ввода-вывода, такие как *open()*. Однако этот подход порождает несколько проблем.

Во-первых, пакет практически не контролирует текущий рабочий каталог

интерпретатора. Поэтому любые операции ввода-вывода должны быть прописаны с использованием абсолютных имён файлов. Поскольку каждый модуль включает переменную `_file_`, содержащую полный путь, определить местоположение не невозможно, но это грязный код.

Во-вторых, пакеты часто инсталлируются как файлы `.zip` или `.egg`, которые не сохраняют файлы так же, как обычные каталоги файловой системы. Поэтому если вы попробуете использовать `open()` на находящемся в архиве файле с данными, это не сработает.

Функция `pkgutil.get_data()` — это высокоуровневый инструмент для получения файла с данными, независимо от того, где или как установлен пакет. Она «просто работает» и возвращает содержимое файла в форме байтовой строки.

Первый аргумент `get_data()` — это строка, которая содержит имя пакета. Вы можете либо предоставить её напрямую, либо использовать специальную переменную, такую как `_package_`. Второй аргумент — это относительное имя файла внутри пакета. При необходимости вы можете переходить по каталогам, используя стандартные пути Unix — до тех пор, пока конечный каталог всё ещё расположен внутри пакета.

10.9. Добавление каталогов в `sys.path`

Задача

У вас есть код Python, который не импортируется, потому что он не размещён в одном из каталогов, перечисленных в `sys.path`. Вы хотели бы добавить новые каталоги в путь поиска Python, но не хотите «зашивать» их в свой код.

Решение

Есть два распространённых приёма, которые позволяют добавить новые каталоги в `sys.path`. Во-первых, вы можете добавить их через переменную окружения `PYTHONPATH`. Например:

```
bash % env PYTHONPATH=/some/dir:/other/dir python3
Python 3.3.0 (default, Oct 4 2012, 10:17:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import sys
>>> sys.path
[ '', '/some/dir', '/other/dir', ...]
>>>
```

В вашем собственном приложении эту переменную можно менять при запуске программы или через скрипт программной оболочки.

Второй подход — создать файл *.pth*, в котором перечисляются каталоги:

```
# myapplication.pth
/some/dir
/other/dir
```

Этот файл *.pth* нужно поместить в один из каталогов *site-packages* Python, которые обычно расположены примерно тут: */usr/local/lib/python3.3/site-packages* или *~/.local/lib/python3.3/site-packages*. При запуске интерпретатора перечисленные в этом файле каталоги будут добавлены в *sys.path*, если они присутствуют в файловой системе. Добавление файлов *.pth* может потребовать доступа на уровне администратора системы, если файлы добавляются к системному интерпретатору Python.

Обсуждение

Столкнувшись с проблемой определения местоположения файлов, вы можете склоняться к написанию кода, который вручную правит значение *sys.path*. Например:

```
import sys
sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

Хотя это «работает», но на практике порождает крайне хрупкий код. Этого нужно избегать. Недостаток этого подхода в том, что он жёстко прописывает имена каталогов в исходном коде. Это может вызывать проблемы с поддержкой, если ваш код будет перемещен. Обычно гораздо лучше конфигурировать пути способом, который можно настраивать, не копаясь в исходниках.

Иногда вы можете обойти проблему жёсткого прописывания каталогов, если аккуратно сконструируете подходящий абсолютный путь, используя переменные уровня модуля, такие как *__file__*. Например:

```
import sys
from os.path import abspath, join, dirname
sys.path.insert(0, abspath(dirname('__file__')), 'src'))
```

Это добавляет каталог `src` в путь поиска таким образом, что этот каталог оказывается размещён в том же каталоге, что и код, который выполняет шаг вставки.

В каталоги `site-packages` обычно инсталлируются сторонние модули и пакеты. Если ваш код был установлен таким образом, то он будет находиться там. Хотя файлы `.pth` для конфигурирования пути поиска должны быть размещены в `site-packages`, они могут ссылаться на любые каталоги в системе. Таким образом вы можете поместить ваш код в совершенно другой набор каталогов, если эти каталоги включены в файл `.pth`.

10.10. Импортирование модулей с использованием передаваемого в форме строки имени

Задача

У вас есть имя модуля, который вы хотите импортировать, но оно хранится в строке. Вы хотели бы вызвать команду `import` с этой строкой.

Решение

Используйте функцию `importlib.import_module()`, чтобы вручную импортировать модуль или часть пакета, имена которых передаются в форме строк. Например:

```
>>> import importlib
>>> math = importlib.import_module('math')
>>> math.sin(2)
0.9092974268256817
>>> mod = importlib.import_module('urllib.request')
>>> u = mod.urlopen('http://www.python.org')
>>>
```

`import_module` просто выполняет те же шаги, что и `import`, но возвращает

получившийся объект модуля обратно в результае. Вам нужно просто сохранить его в переменной и затем использовать, как обычный модуль.

Если вы работаете с пакетами, `import_module()` также может быть использована для выполнения относительного импортирования. Однако вам нужно предоставить ей дополнительный аргумент. Например:

```
import importlib

# Same as 'from . import b'
b = importlib.import_module('.b', __package__)
```

Обсуждение

Задача ручного импортирования модулей с использованием `import_module()` чаще всего возникает при написании кода, который как-то манипулирует модулями или создает над ними обёртки. Например, если вы реализовываете какой-то собственный механизм импортирования, который подразумевает загрузку модуля по имени и применение патчей к загруженному коду.

В старом коде вы иногда сможете встретить использование встроенной функции `import()` для выполнения импортирования. Хотя это и работает, `importlib.import_module()` обычно проще в использовании.

См. рецепт 10.11., где приведён продвинутый пример кастомизации процесса импортирования.

10.11. Загрузка модулей с удалённого компьютера с использованием хуков импортирования

Задача

Вы хотите изменить инструкцию импортирования Python, чтобы она могла прозрачно загружать модули с удалённого компьютера.

Решение

Во-первых, сразу отметим важность соблюдения правил безопасности. Идея,

которая обсуждается в этом рецепте, станет ужасной без некого дополнительного слоя аутентификации и безопасности. Цель этого рецепта — глубоко погрузиться во внутренний механизм работы инструкции *import*. Если вы заставите этот рецепт работать и поймете внутреннюю механику, то получите серьёзный фундамент знаний для приспособления *import* под практически любые нужды. Имея это в виду, давайте продолжим!

В сердце этого рецепта лежит желание расширить функциональность инструкции *import*. Есть несколько подходов к тому, как можно это сделать, но для наглядности начнём с создания такого каталога с кодом Python:

```
testcode/
    spam.py
    fib.py
    grok/
        __init__.py
        blah.py
```

Содержимое этих файлов не имеет значения, но поместим в каждый из них несколько простых инструкций и функций, чтобы вы могли их потестировать и посмотреть на получающийся при импортировании вывод. Например:

```
# spam.py
print("I'm spam")

def hello(name):
    print('Hello %s' % name)

# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# grok/__init__.py
print("I'm grok.__init__")
# grok/blah.py
print("I'm grok.blah")
```

Наша цель — разрешить удалённый доступ к этим файлам, как к модулям. Вероятно, наиболее простой способ сделать это — опубликовать их на веб-сервере. Просто зайдите в каталог *testcode* и запустите Python таким

образом:

```
bash % cd testcode
bash % python3 -m http.server 15000
Serving HTTP on 0.0.0.0 port 15000 ...
```

Оставим этот сервер в запущенном состоянии и отдельно запустим интерпретатор Python. Убедитесь, что вы имеете доступ к удалённым файлам, используя *urllib*. Например:

```
>>> from urllib.request import urlopen
>>> u = urlopen('http://localhost:15000/fib.py')
>>> data = u.read().decode('utf-8')
>>> print(data)
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

Загрузка исходного кода с сервера — это база для оставшейся части этого рецепта. Если конкретно, то вместо ручного получения файла с исходным кодом с использованием *urlopen()*, мы переделаем инструкцию *import* так, что она будет делать это «за кулисами».

Первый подход к загрузке удалённого модуля — создать явную функцию загрузки, которая будет этим заниматься. Например:

```
import imp
import urllib.request
import sys

def load_module(url):
    u = urllib.request.urlopen(url)
    source = u.read().decode('utf-8')
    mod = sys.modules.setdefault(url, imp.new_module(url))
    code = compile(source, url, 'exec')
    mod.__file__ = url
    mod.__package__ = ''
    exec(code, mod.__dict__)
    return mod
```

Эта функция просто загружает исходный код, компилирует его в объект кода, используя `compile()`, а затем выполняет словарь созданного объекта модуля. Вот как вы могли бы использовать эту функцию:

```
>>> fib = load_module('http://localhost:15000/fib.py')
I'm fib
>>> fib.fib(10)
89
>>> spam = load_module('http://localhost:15000/spam.py')
I'm spam
>>> spam.hello('Guido')
Hello Guido
>>> fib
<module 'http://localhost:15000/fib.py' from 'http://localhost:15000/fib.py'>
>>> spam
<module 'http://localhost:15000/spam.py' from 'http://localhost:15000/spam.py'>
>>>
```

Как вы можете видеть, это работает для простых модулей. Однако функция не подключена к обычной инструкции `import`, а расширение кода для поддержки более продвинутых конструкций, таких как пакеты, потребует дополнительной работы.

Намного более ловкий подход — создать собственный импортировщик. Первый способ это сделать — это написать так называемый «импортировщик мета-пути». Вот пример:

```
# urlimport.py

import sys
import importlib.abc
import imp
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from html.parser import HTMLParser

# Отладка
import logging
log = logging.getLogger(__name__)

# Получение ссылок из переданного URL
def _get_links(url):
    class LinkParser(HTMLParser):
        def handle_starttag(self, tag, attrs):
            if tag == 'a':
                attrs = dict(attrs)
```

```

        links.add(attrs.get('href').rstrip('/'))
links = set()
try:
    log.debug('Getting links from %s' % url)
    u = urlopen(url)
    parser = LinkParser()
    parser.feed(u.read().decode('utf-8'))
except Exception as e:
    log.debug('Could not get links. %s', e)
log.debug('links: %r', links)
return links

class UrlMetaFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._links = { }
        self._loaders = { baseurl : UrlModuleLoader(baseurl) }

    def find_module(self, fullname, path=None):
        log.debug('find_module: fullname=%r, path=%r', fullname, path)
        if path is None:
            baseurl = self._baseurl
        else:
            if not path[0].startswith(self._baseurl):
                return None
            baseurl = path[0]

        parts = fullname.split('.')
        basename = parts[-1]
        log.debug('find_module: baseurl=%r, basename=%r', baseurl, basename)

        # Проверка кэша ссылок
        if basename not in self._links:
            self._links[baseurl] = _get_links(baseurl)

        # Проверка того, что это не пакет
        if basename in self._links[baseurl]:
            log.debug('find_module: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Попытка загрузить пакет (которая обращается к __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                self._links[fullurl] = _get_links(fullurl)
                self._loaders[fullurl] = UrlModuleLoader(fullurl)
                log.debug('find_module: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_module: package failed. %s', e)
                loader = None
            return loader

```

```
# Обычный модуль
filename = basename + '.py'
if filename in self._links[baseurl]:
    log.debug('find_module: module %r found', fullname)
    return self._loaders[baseurl]
else:
    log.debug('find_module: module %r not found', fullname)
    return None

def invalidate_caches(self):
    log.debug('invalidating link cache')
    self._links.clear()

# Загрузчик модуля из URL
class UrlModuleLoader(importlib.abc.SourceLoader):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._source_cache = {}

    def module_repr(self, module):
        return '<urlmodule %r from %r>' % (module.__name__, module.__file__)

    # Требуемый метод
    def load_module(self, fullname):
        code = self.get_code(fullname)
        mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
        mod.__file__ = self.get_filename(fullname)
        mod.__loader__ = self
        mod.__package__ = fullname.rpartition('.')[0]
        exec(code, mod.__dict__)
        return mod

    # Необязательные расширения
    def get_code(self, fullname):
        src = self.get_source(fullname)
        return compile(src, self.get_filename(fullname), 'exec')

    def get_data(self, path):
        pass

    def get_filename(self, fullname):
        return self._baseurl + '/' + fullname.split('.')[-1] + '.py'

    def get_source(self, fullname):
        filename = self.get_filename(fullname)
        log.debug('loader: reading %r', filename)
        if filename in self._source_cache:
            log.debug('loader: cached %r', filename)
            return self._source_cache[filename]
```

```

try:
    u = urlopen(filename)
    source = u.read().decode('utf-8')
    log.debug('loader: %r loaded', filename)
    self._source_cache[filename] = source
    return source
except (HTTPError, URLError) as e:
    log.debug('loader: %r failed. %s', filename, e)
    raise ImportError("Can't load %s" % filename)

def is_package(self, fullname):
    return False

# Загрузчик пакета из URL
class UrlPackageLoader(UrlModuleLoader):
    def load_module(self, fullname):
        mod = super().load_module(fullname)
        mod.__path__ = [ self._baseurl ]
        mod.__package__ = fullname

    def get_filename(self, fullname):
        return self._baseurl + '/' + '__init__.py'

    def is_package(self, fullname):
        return True

# Вспомогательные функции для инсталляции/деинсталляции загрузчика
_installed_meta_cache = { }
def install_meta(address):
    if address not in _installed_meta_cache:
        finder = UrlMetaFinder(address)
        _installed_meta_cache[address] = finder
        sys.meta_path.append(finder)
        log.debug('%r installed on sys.meta_path', finder)

def remove_meta(address):
    if address in _installed_meta_cache:
        finder = _installed_meta_cache.pop(address)
        sys.meta_path.remove(finder)
        log.debug('%r removed from sys.meta_path', finder)

```

Вот пример интерактивного сеанса, показывающего, как использовать приведённый выше код:

```

>>> # сейчас импортирование не удаётся
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

```

```
>>> # Загружаем импортировщик и пробуем ещё раз (всё работает)
>>> import urlimport
>>> urlimport.install_meta('http://localhost:15000')
>>> import fib
I'm fib
>>> import spam
I'm spam
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

Это конкретное решение включает присваивание последней записи в `sys.meta_path` экземпляра специального объекта-поисковика `UrlMetaFinder`. Когда модули загружаются, происходит консультация с поисковиками в `sys.meta_path`, чтобы определить местоположение модуля. В этом примере экземпляр `UrlMetaFinder` становится «поисковиком последнего шанса», который запускается, когда модуль не найден ни в одном из обычных местоположений.

Для обобщения подхода к реализации класс `UrlMetaFinder` обёрнут вокруг определяемого пользователем URL. Внутри наш поисковик строит множество валидных ссылок путём получения их из переданного URL. Когда импортирование произведено, имя модуля сравнивается с этим множеством известных ссылок. Если найдено совпадение, используется отдельный класс `UrlModuleLoader`, который загружает исходный код с удалённого компьютера и в результате создает объект модуля. Ссылки кэшируются, чтобы избежать ненужных HTTP-запросов при повторяющихся операциях импортирования.

Второй подход к кастомизации импортирования — написать хук, который прикрепляется напрямую к переменной `sys.path`, распознавая некоторые шаблоны наименования каталогов. Добавьте приведённый ниже класс и поддерживающие функции в `urlimport.py`:

```
# urlimport.py

# ... включите предыдущий код здесь ...

# Класс для поиска пути в URL
class UrlPathFinder(importlib.abc.PathEntryFinder):
    def __init__(self, baseurl):
        self._links = None
```

```
self._loader = UrlModuleLoader(baseurl)
self._baseurl = baseurl

def find_loader(self, fullname):
    log.debug('find_loader: %r', fullname)
    parts = fullname.split('.')
    basename = parts[-1]
    # Check link cache
    if self._links is None:
        self._links = []          # См. обсуждение
        self._links = _get_links(self._baseurl)

    # Проверить, пакет ли это
    if basename in self._links:
        log.debug('find_loader: trying package %r', fullname)
        fullurl = self._baseurl + '/' + basename
        # Попытка загрузить пакет (которая обращается к __init__.py)
        loader = UrlPackageLoader(fullurl)
        try:
            loader.load_module(fullname)
            log.debug('find_loader: package %r loaded', fullname)
        except ImportError as e:
            log.debug('find_loader: %r is a namespace package', fullurl)
            loader = None
        return (loader, [fullurl])

    # Обычный модуль
    filename = basename + '.py'
    if filename in self._links:
        log.debug('find_loader: module %r found', fullname)
        return (self._loader, [])
    else:
        log.debug('find_loader: module %r not found', fullname)
        return (None, [])

def invalidate_caches(self):
    log.debug('invalidating link cache')
    self._links = None

# Проверка пути на предмет того, выглядит ли он как URL
_url_path_cache = {}
def handle_url(path):
    if path.startswith(('http://', 'https://')):
        log.debug('Handle path? %s. [Yes]', path)
        if path in _url_path_cache:
            finder = _url_path_cache[path]
        else:
            finder = UrlPathFinder(path)
            _url_path_cache[path] = finder
    return finder
else:
```

```
    log.debug('Handle path? %s. [No]', path)

def install_path_hook():
    sys.path_hooks.append(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Installing handle_url')

def remove_path_hook():
    sys.path_hooks.remove(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Removing handle_url')
```

Чтобы использовать этот основанный на пути поисковик, просто добавьте URLs в `sys.path`. Например:

```
>>> # Initial import fails
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Установка хука пути
>>> import urlimport
>>> urlimport.install_path_hook()

>>> # Импортирование всё ещё не удаётся (не в пути)
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Добавляем запись в sys.path и наблюдаем, как всё отлично работает
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
I'm fib
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

Ключ к этому последнему примеру заключается в функции `handle_url()`, которая добавлена в переменную `sys.path_hooks`. Когда обрабатываются записи `sys.path`, вызываются функции из `sys.path_hooks`. Если какая-либо из этих функций возвращает объект поисковика, этот поисковик используется

для попытки загрузки модулей для этой записи в `sys.path`.

Стоит отметить, что удалённо импортируемые модули работают точно так же, как и любые другие. Например:

```
>>> fib
<urlmodule 'fib' from 'http://localhost:15000/fib.py'>
>>> fib.__name__
'fib'
>>> fib.__file__
'http://localhost:15000/fib.py'
>>> import inspect
>>> print(inspect.getsource(fib))
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

Обсуждение

Перед детальным обсуждением этого рецепта, стоит отметить, что модули, пакеты и механизм импорта Python — это одна из сложнейших частей всего языка. Часто даже самые опытные программисты плохо её понимают — до тех пор, пока не вложат некоторое количество усилий, чтобы погрузиться в тему. Есть несколько важнейших документов, которые достойны прочтения, включая документацию к [importlib](#) и [PEP 302](#). Эту документацию мы не будем повторять тут, но некоторые важнейшие моменты обсудим.

Во-первых, если вы хотите создать новый объект модуля, вы используете функцию `imp.new_module()`. Например:

```
>>> import imp
>>> m = imp.new_module('spam')
>>> m
<module 'spam'>
>>> m.__name__
'spam'
>>>
```

Объекты модулей обычно имеют несколько ожидаемых атрибутов, включая

`__file__` (имя файла, из которого был загружен модуль) и `__package__` (имя пакета, в котором находится модуль, если он есть).

Во-вторых, модули кэшируются интерпретатором. Закэшированный модуль может быть найден в словаре `sys.modules`. Из-за этого кэширования распространённой практикой является объединение кэширования и создания модуля в один этап. Например:

```
>>> import sys
>>> import imp
>>> m = sys.modules.setdefault('spam', imp.new_module('spam'))
>>> m
<module 'spam'>
>>>
```

Главная причина делать это в том, что если модуль с заданным именем уже существует, то вы вместо пересоздания получите уже созданный модуль. Например:

```
>>> import math
>>> m = sys.modules.setdefault('math', imp.new_module('math'))
>>> m
<module 'math' from '/usr/local/lib/python3.3/lib-dynload/math.so'>
>>> m.sin(2)
0.9092974268256817
>>> m.cos(2)
-0.4161468365471424
>>>
```

Поскольку создавать модули легко, то можно без труда написать простые функции, такие как `load_module()` из первой части этого рецепта. Недостаток этого подхода в том, что с его помощью трудно обрабатывать более сложные случаи, такие как импортирование пакетов. Чтобы работать с пакетами, вам придётся заново реализовать большую часть «подкапотной» логики, которая является частью обычной инструкции `import` (например, проверку каталогов, поиск файлов `__init__.py`, выполнение этих файлов, установка путей и т.д.) Эта высокая сложность — одна из причин, из-за которых часто лучше расширить инструкцию `import` напрямую, а не определять собственную функцию.

Расширение инструкции `import` выполняется прямолинейно, но вовлекает достаточно много «движущихся частей». На высшем уровне операции `import` обрабатываются списком поисковиков «мета-путей», который вы можете найти в списке `sys.meta_path`. Если вы выведете его значение, то увидите

следующее:

```
>>> from pprint import pprint
>>> pprint(sys.meta_path)
[<class '_frozen_importlib.BuiltinImporter'>,
 <class '_frozen_importlib.FrozenImporter'>,
 <class '_frozen_importlib.PathFinder'>]
>>>
```

При выполнении инструкции типа *import fib*, интерпретатор проходит по объектам-поисковикам из *sys.meta_path* и вызывает их метод *find_module()*, чтобы найти подходящий загрузчик модуля. Определите нижеприведённый класс и попробуйте поэкспериментировать с ними, чтобы разобраться в том, как это работает:

```
>>> class Finder:
...     def find_module(self, fullname, path):
...         print('Looking for', fullname, path)
...         return None
...
>>> import sys
>>> sys.meta_path.insert(0, Finder()) # Вставить в начало
>>> import math
Looking for math None
>>> import types
Looking for types None
>>> import threading
Looking for threading None
Looking for time None
Looking for traceback None
Looking for linecache None
Looking for tokenize None
Looking for token None
>>>
```

Обратите внимание на то, что метод *find_module()* запускается при каждом импортировании. Роль аргумента *path* здесь заключается в обработке пакетов. Когда пакеты импортированы, это список каталогов, которые находятся в атрибуте пакета *__path__*. Это пути, которые нужно проверить, чтобы найти подкомпоненты пакета. Например, посмотрите установки пути для *xml.etree* и *xml.etree.ElementTree*:

```
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml.etree ['/usr/local/lib/python3.3/xml']
```

```
Looking for xml.etree.ElementTree ['/usr/local/lib/python3.3/xml/etree']
Looking for warnings None
Looking for contextlib None
Looking for xml.etree.ElementPath ['/usr/local/lib/python3.3/xml/etree']
Looking for _elementtree None
Looking for copy None
Looking for org None
Looking for pyexpat None
Looking for ElementC14N None
```

Положение поисковика в `sys.meta_path` является критически важным.

Переместите его из начала в конец списка и попробуйте выполнить несколько операций импортирования:

```
>>> del sys.meta_path[0]
>>> sys.meta_path.append(Finder())
>>> import urllib.request
>>> import datetime
```

Теперь вы не видите никакого вывода, потому что импортирование обрабатывается другими записями из `sys.meta_path`. В этом случае вы увидите их только в том случае, если они включатся по причине попытке импортирования несуществующих модулей:

```
>>> import fib
Looking for fib None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import xml.superfast
Looking for xml.superfast ['/usr/local/lib/python3.3/xml']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'xml.superfast'
>>>
```

Тот факт, что вы можете инсталлировать поисковик с целью «отлова» неизвестных модулей, является ключом к пониманию класса `UrlMetaFinder` в этом рецепте. Экземпляр `UrlMetaFinder` добавлен в конец `sys.meta_path`, где служит своего рода «импортировщиком последнего шанса». Если запрошенное имя модуля не может быть обнаружено никаким другим механизмом импортирования, оно обрабатывается этим поисковиком. При обработке пакетов нужно принять некоторые предосторожности. Если конкретно, то представленное в аргументе `path` значение должно быть

проверено, чтобы посмотреть, не начинается ли оно с зарегистрированного в поисковике URL. Если нет, подмодуль принадлежит какому-то другому поисковику и должен быть проигнорирован.

Дополнительная обработка пакетов производится с помощью класса *UrlPackageLoader*. Этот класс, вместо импортирования имени пакета, пытается загрузить файл *__init__.py*. Он также устанавливает атрибут модуля *__path__*. Эта последняя часть является критически важной, поскольку установленное значение будет передано в последующие вызовы *find_module()* при загрузке подмодулей пакета.

Основанный на путях импортирования расширяет эти идеи, но базируется на другом механизме. Как вы знаете, *sys.path* — это список каталогов, где Python ищет модули. Например:

```
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
[ '',
  '/usr/local/lib/python33.zip',
  '/usr/local/lib/python3.3',
  '/usr/local/lib/python3.3/plat-darwin',
  '/usr/local/lib/python3.3/lib-dynload',
  '/usr/local/lib/python3.3/site-packages']
>>>
```

Каждая запись в *sys.path* дополнительно прикреплена к объекту поисковика. Вы можете просматривать эти поисковики путём обращения к *sys.path_importer_cache*:

```
>>> pprint(sys.path_importer_cache)
{'.': FileFinder('.'),
 '/usr/local/lib/python3.3': FileFinder('/usr/local/lib/python3.3'),
 '/usr/local/lib/python3.3/': FileFinder('/usr/local/lib/python3.3/'),
 '/usr/local/lib/python3.3/collections': FileFinder('...python3.3/collec',
 '/usr/local/lib/python3.3/encodings': FileFinder('...python3.3/encodin',
 '/usr/local/lib/python3.3/lib-dynload': FileFinder('...python3.3/lib-dj',
 '/usr/local/lib/python3.3/plat-darwin': FileFinder('...python3.3/plat-c',
 '/usr/local/lib/python3.3/site-packages': FileFinder('...python3.3/site',
 '/usr/local/lib/python33.zip': None}
>>>
```

sys.path_importer_cache обычно намного больше *sys.path*, потому что в него записываются поисковики для всех известных каталогов, из которых

загружается код. Это включает подкаталоги пакетов, которые обычно не включаются в `sys.path`.

Чтобы выполнить `import fib`, каталоги в `sys.path` проверяются по порядку. Для каждого каталога имя `fib` представляется ассоциированному поисковику, найденному в `sys.path_importer_cache`. Вам стоит исследовать этот аспект путём создания собственного поисковика и помещения записи в кэш.

Попробуйте провести такой эксперимент:

```
>>> class Finder:
...     def find_loader(self, name):
...         print('Looking for', name)
...         return (None, [])
...
>>> import sys
>>> Добавить запись "debug" в кэш импортировщика
>>> sys.path_importer_cache['debug'] = Finder()
>>> # Добавить каталог "debug" в sys.path
>>> sys.path.insert(0, 'debug')
>>> import threading
Looking for threading
Looking for time
Looking for traceback
Looking for linecache
Looking for tokenize
Looking for token
>>>
```

Здесь вы установили новую запись кэша для имени `debug` и установили имя `debug` в качестве первой записи в `sys.path`. В последующих операциях импортирования вы видите, как запускается ваш поисковик. Однако, поскольку он возвращает `(None, [])`, процесс обработки просто переходит к следующей записи.

Наполнение `sys.path_importer_cache` контролируется списком функций, который хранится в `sys.path_hooks`. Попробуйте провести эксперимент, в котором вы очистите кэш и добавите новую функцию проверки пути в `sys.path_hooks`:

```
>>> sys.path_importer_cache.clear()
>>> def check_path(path):
...     print('Checking', path)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
```

```
>>> import fib
Checked debug
Checking .
Checking /usr/local/lib/python33.zip
Checking /usr/local/lib/python3.3
Checking /usr/local/lib/python3.3/plat-darwin
Checking /usr/local/lib/python3.3/lib-dynload
Checking /Users/beazley/.local/lib/python3.3/site-packages
Checking /usr/local/lib/python3.3/site-packages
Looking for fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>>
```

Как вы можете видеть, функция `check_path()` вызывается для каждой записи из `sys.path`. Однако, поскольку возбуждается исключение `ImportError`, ничего другого не происходит (проверка просто переходит к следующей функции в `sys.path_hooks`).

Используя свои знания о том, как обрабатывается `sys.path`, вы можете инсталлировать кастомную функцию проверки пути, которая ищет шаблоны имён файлов, такие как URLs. Например:

```
>>> def check_url(path):
...     if path.startswith('http://'):
...         return Finder()
...     else:
...         raise ImportError()
...
>>> sys.path.append('http://localhost:15000')
>>> sys.path_hooks[0] = check_url
>>> import fib
Looking for fib
# Вывод поисковика!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Обратите внимание на установку поисковика в sys.path_importer_cache
>>> sys.path_importer_cache['http://localhost:15000']
<__main__.Finder object at 0x10064c850>
>>>
```

Это ключевой механизм работы последней части данного рецепта. По сути, кастомная функция проверки пути была установлена, чтобы искать URLs в `sys.path`. Когда они встречаются, создается новый экземпляр `UrlPathFinder` и

инсталлируется в `sys.path_importer_cache`. Начиная с этого момента, все инструкции импортирования, которые проходят через эту часть `sys.path`, будут пытаться использовать ваш кастомный поисковик.

Работа с пакетами с помощью основанного на путях импортировщика достаточно сложна и опирается на возвращаемое значение метода `find_loader()`. Для простых модулей `find_loader()` возвращает кортеж (`loader, None`), где `loader` — это экземпляр загрузчика, который будет импортировать модуль.

Для обычного пакета `find_loader()` возвращает кортеж (`loader, path`), где `loader` — это экземпляр загрузчика, который будет импортировать пакет (и выполнять `__init__.py`), а `path` — это список каталогов, которые составят начальное наполнение атрибута пакета `__path__`. Например, если базовый URL был <http://localhost:15000>, а пользователь выполнил `import grok`, возвращенный `find_loader()` путь будет таким: ['<http://localhost:15000/grok>'].

`find_loader()` должна дополнительно учитывать возможность столкнуться с пакетом пространства имён. Это пакет, где присутствуют валидные имена каталогов пакета, но отсутствуют файлы `__init__.py`. Для этого случая `find_loader()` должна возвращать кортеж (`None, path`), где `path` — это список каталогов, которые составили бы атрибут пакета `__file__`, определённый файлом `__init__.py` (path is a list of directories that would have made up the package's `__path__` attribute had it defined an `__init__.py` file). В этом случае механизм импортирования переходит дальше, чтобы проверить следующие каталоги в `sys.path`. Если будут найдены дополнительные пакеты пространств имён, все получившиеся пути соединяются вместе, чтобы образовать финальный пакет пространств имён. См. [рецепт 10.5.](#), где приведена дополнительная информация о пакетах пространств имён.

В обработке пакетов есть элемент рекурсии, что не очевидно ни в решении, ни в работе. Все пакеты содержат внутреннюю настройку путей, которая может быть найдена в атрибуте `__path__`. Например:

```
>>> import xml.etree.ElementTree
>>> xml.__path__
['/usr/local/lib/python3.3/xml']
>>> xml.etree.__path__
['/usr/local/lib/python3.3/xml/etree']
>>>
```

Как было упомянуто, установка `__path__` контролируется путём возвращаемого значения метода `find_loader()`. Однако последующая обработка `__path__` также производится функциями из `sys.path_hooks`. Когда подкомпоненты пакета загружаются, записи в `__path__` проверяются функцией `handle_url()`. Это вызывает создание новых экземпляров `UrlPathFinder` и добавление их в `sys.path_importer_cache`.

Оставшаяся сложная часть этой реализации касается поведения функции `handle_url()` и её взаимодействия с используемой внутри функцией `_get_links()`. Если ваша реализация поисковика использует другие модули (например, `urllib.request`), существует вероятность того, что эти модули предпримут попытки импортирования в середине выполнения операции поисковика. Это может вызвать рекурсивный цикл выполнения `handle_url()` и других частей поисковика. Чтобы учёсть такую возможность, реализация поддерживает кэш созданных поисковиков (по одному на URL). Это позволяет избежать проблемы с созданием дублирующихся поисковиков. Следующий фрагмент кода удостоверяет, что поисковик не отвечает ни на какие запросы импортирования, пока он находится в процессе получения начального набора ссылок:

```
# Проверяем кэш ссылок
if self._links is None:
    self._links = []                      # См. обсуждение
    self._links = _get_links(self._baseurl)
```

Эта проверка может и не понадобиться вам в других реализациях, но для этого примера, использующего URLs, она необходима.

И, наконец, метод `invalidate_caches()` обоих поисковиков — это вспомогательный метод, который должен очищать внутренние кэши при изменении исходного кода. Этот метод вызывается, когда пользователь вызывает `importlib.invalidate_caches()`. Вы можете использовать его, если хотите, чтобы импортировщики URL перечитали список ссылок — возможно, чтобы получить доступ к добавленным файлам.

В сравнении двух подходов (изменении `sys.meta_path` и использовании хуков пути) помогает «высокоуровневый взгляд». Импортировщики, установленные с использованием `sys.meta_path`, свободны обрабатывать модули так, как пожелают. Например, они могут загружать модули из базы данных или импортировать их способом, который радикально отличается от стандартного процесса для модулей/пакетов. Эта свобода также означает,

что импортировщики должны вести больше «бухгалтерии» и заниматься внутренним управлением. Это объясняет, например, почему реализация *UrlMetaFinder* требует собственного кэширования ссылок, загрузчиков и прочих деталей. С другой стороны, основанные на путях хуки сильнее привязаны к обработке *sys.path*. По причине связи с *sys.path*, модули, загружаемые с помощью таких расширений, будут склонны иметь те же возможности, что и обычные модули и пакеты, к которым привыкли программисты.

Предполагая, что к этому моменту у вас еще не взорвалась голова, скажем, что ключом к пониманию и экспериментированию с этим рецептом может быть добавление вызовов логирования. Вы можете включить логирование и попробовать что-то такое:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import urlimport
>>> urlimport.install_path_hook()
DEBUG:urlimport:Installing handle_url
>>> import fib
DEBUG:urlimport:Handle path? /usr/local/lib/python33.zip. [No]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
DEBUG:urlimport:Handle path? http://localhost:15000. [Yes]
DEBUG:urlimport:Getting links from http://localhost:15000
DEBUG:urlimport:links: {'spam.py', 'fib.py', 'grok'}
DEBUG:urlimport:find_loader: 'fib'
DEBUG:urlimport:find_loader: module 'fib' found
DEBUG:urlimport:loader: reading 'http://localhost:15000/fib.py'
DEBUG:urlimport:loader: 'http://localhost:15000/fib.py' loaded
I'm fib
>>>
```

И последнее: рекомендуем провести некоторое время в медитации над [PEP 302](#) и документации к *importlib*.

10.12. Применение к модулям патчей во время импортирования

Задача

Вы хотите пропатчить функции в существующем модуле или применить к ним декораторы. Однако вы хотите сделать это только в том случае, когда модуль импортируется и где-то используется.

Решение

Основная проблема тут в том, что вы хотите выполнять действия в ответ на загрузку модуля. Возможно, вы хотите запускать некую функцию обратного вызова (коллбэк), которая будет уведомлять вас о загрузке модуля.

Эта задача может быть решена с помощью механизма хуков импортирования, который мы обсудили в [рецепте 10.11](#). Вот возможное решение:

```
# postimport.py

import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path=None):
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        importlib.import_module(fullname)
        module = sys.modules[fullname]
        for func in _post_import_hooks[fullname]:
            func(module)
        self._finder._skip.remove(fullname)
        return module
```

```

def when_imported(fullname):
    def decorate(func):
        if fullname in sys.modules:
            func(sys.modules[fullname])
        else:
            _post_import_hooks[fullname].append(func)
        return func
    return decorate

sys.meta_path.insert(0, PostImportFinder())

```

Чтобы использовать этот код, примените декоратор `when_imported()`.

Например:

```

>>> from postimport import when_imported
>>> @when_imported('threading')
... def warn_threads(mod):
...     print('Threads? Are you crazy?')
...
>>>
>>> import threading
Threads? Are you crazy?
>>>

```

Более практический пример — применение декоратора к существующим определениям:

```

from functools import wraps
from postimport import when_imported

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper

# Пример
@when_imported('math')
def add_logging(mod):
    mod.cos = logged(mod.cos)
    mod.sin = logged(mod.sin)

```

Обсуждение

Этот рецепт опирается на хуки импортирования, которые обсуждаются в

[рецепте 10.11.](#), но с небольшим вывертом.

Во-первых, роль декоратора `@when_imported` состоит в том, чтобы зарегистрировать функции-обработчики, которые запускаются при импортировании. Декоратор проверяет `sys.modules`, чтобы посмотреть, не загружен ли уже модуль. Если это так, обработчик вызывается сразу же. Иначе обработчик будет добавлен в список в словаре `_post_import_hooks`. Назначение `_post_import_hooks` в том, чтобы просто собрать все объекты-обработчики, которые зарегистрированы для каждого модуля. В принципе, для передаваемого модуля может быть зарегистрировано больше одного обработчика.

Чтобы запустить подготовленные действия в `_post_import_hooks` после импортирования модуля, класс `PostImportFinder` инсталлируется в качестве первого элемента в `sys.meta_path`. Как вы помните по [рецепту 10.12.](#), `sys.meta_path` содержит список объектов-поисковиков, к которым по порядку происходят обращения, чтобы найти модули. Путём установки `PostImportFinder` в качестве первого элемента мы «отлавливаем» все случаи импортирования модулей.

В этом рецепте, однако, роль `PostImportFinder` заключается не в загрузке модулей, а в запуске нужных действий по завершению импортирования. Чтобы это сделать, импортование делегируется другим поисковикам в `sys.meta_path`. Вместо того, чтобы пытаться сделать это напрямую, в классе `PostImportLoader` рекурсивно вызывается функция `imp.import_module()`. Чтобы избежать застревания в бесконечном цикле, `PostImportFinder` хранит набор (множество) всех модулей, которые в настоящий момент находятся в процессе загрузки. Если имя модуля присутствует в этом списке, оно просто игнорируется `PostImportFinder`. Это вызывает передачу запроса на импортование другому поисковику из `sys.meta_path`.

После того, как модуль был загружен с помощью `imp.import_module()`, все обработчики, в настоящий момент зарегистрированные в `post_import_hooks`, вызываются с новым загруженным модулем в качестве аргумента. Начиная с этой точки, обработчики свободны делать с модулем всё, что пожелают.

Основная фишка показанного в этом рецепте подхода — то, что патчинг модуля происходит бесшовно, несмотря на то, где или как рассматриваемый модуль загружается. Вы просто пишете функцию-обработчик, декорируете ее с помощью `@when_imported()`, и всё магическим образом работает.

Предостережение: этот рецепт не работает для модулей, которые явно загружаются с помощью `imp.reload()`. Так что если вы перезагружаете ранее загруженный модуль, «послеимпортная» функция-обработчик не запускается заново (ещё один довод не использовать `reload()` в продакшн-коде). С другой стороны, если вы удалите модуль из `sys.modules` и повторите импорт, вы увидите, что обработчик запускается.

Дополнительную информацию о послеимпортных хуках вы найдёте в [PEP 369](#). На момент написания этой книги данный PEP был отозван автором, поскольку он устарел по отношению к текущей реализации модуля `importlib`. Однако с помощью этого рецепта вы можете достаточно легко создать собственную реализацию.

10.13. Установка пакетов «только для себя»

Задача

Вы хотите установить сторонний пакет, но у вас нет прав для установки пакетов в системный Python. Или же вы хотите установить пакет только для себя, а не для всех пользователей системы.

Решение

В Python есть инсталляционный каталог для конкретного пользователя, который обычно находится в каталоге типа `~/.local/lib/python3.3/site-packages`. Чтобы заставить пакеты устанавливаться в этот каталог, передайте опцию `--user` в команду установки. Например:

```
python3 setup.py install --user
```

или

```
pip install --user packagename
```

Каталог `site-packages` конкретного пользователя обычно расположен перед системным каталогом `site-packages` в `sys.path`. Поэтому пакеты, которые устанавливаются с использованием этого приёма, имеют приоритет перед

пакетами, уже установленными в системе (хотя это не всегда так, у сторонних менеджеров пакетов типа `distribute` и `pip` поведение может отличаться).

Обсуждение

Обычно пакеты устанавливаются в общесистемный каталог `site-packages` — например, такой: `/usr/local/lib/python3.3/site-packages`. Однако для этого нужны права администратора и команда `sudo`. Даже если права у вас есть, использование `sudo` для установки нового непроверенного пакета может быть не лучшим вариантом.

Установка пакетов в каталог пользователя часто будет эффективным способом, позволяющим создать кастомную инсталляцию.

Вы также можете создать виртуальное окружение, что обсуждается в следующем рецепте.

10.14. Создание нового окружения Python

Задача

Вам нужно создать новое окружение Python, в котором вы хотите устанавливать модули и пакеты. Однако вы хотите сделать это без установки новой копии Python или внесения изменений, которые повлияют на системный Python.

Решение

Вы можете создать новое «виртуальное» окружение, используя команду `pyvenv`. Эта команда установлена в тот же каталог, что и интерпретатор Python — или, на Windows, в каталог `Scripts`. Вот пример:

```
bash % pyvenv Spam  
bash %
```

Имя, предоставленное в `pyvenv`, станет именем нового каталога. После создания каталог `Spam` будет выглядеть как-то так:

```
bash % cd Spam  
bash % ls
```

```
bin      include     lib      pyvenv.cfg
bash %
```

В каталоге *bin* вы найдете интерпретатор Python, который можете использовать. Например:

```
bash % Spam/bin/python3
Python 3.3.0 (default, Oct 6 2012, 15:45:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
[ '',
  '/usr/local/lib/python33.zip',
  '/usr/local/lib/python3.3',
  '/usr/local/lib/python3.3/plat-darwin',
  '/usr/local/lib/python3.3/lib-dynload',
  '/Users/beazley/Spam/lib/python3.3/site-packages' ]
>>>
```

Главная особенность этого интерпретатора в том, что его каталог *site-packages* установлен в качестве нового окружения. Если вы захотите установить сторонние пакеты, они будут установлены сюда, а не в системный каталог *site-packages*.

Обсуждение

Создание виртуального окружения по большей части применяется для установки и управления сторонними пакетами. Как вы можете видеть в примере, переменная *sys.path* содержит каталоги обычного системного Python, но каталог *site-packages* теперь другой.

После создания нового виртуального окружения следующим шагом часто становится установка менеджера пакетов, такого как *distribute* или *pip*. При установке этих инструментов вы должны убедиться, что используете интерпретатор, который является частью виртуального окружения. Так пакеты будут установлены в новый каталог *site-packages*.

Хотя виртуальное окружение может выглядеть как копия инсталляции Python, оно на самом деле состоит всего из нескольких файлов и символьических ссылок. Все файлы стандартной библиотеки и исполняемые файлы интерпретатора приходят из изначальной инсталляции Python. Так что

виртуальные окружения не только просты в создании, но и почти не тратят системных ресурсов.

По умолчанию виртуальные окружения полностью чисты и не содержат сторонних пакетов. Если вы хотите включить уже установленные пакеты в виртуальное окружение, создайте его с опцией `--system-site-packages`.

Например:

```
bash % pyvenv --system-site-packages Spam  
bash %
```

Дополнительную информацию о `pyenv` и виртуальных окружениях вы найдёте в [PEP 405](#).

10.15. Распространение пакетов

Задача

Вы написали полезную библиотеку и хотите передать её другим людям.

Решение

Если вы хотите начать распространение своего кода, первым делом вам нужно присвоить ему уникальное имя и подчистить структуру каталогов. Например, типичный пакет может выглядеть так:

```
projectname/  
  README.txt  
  Doc/  
    documentation.txt  
  projectname/  
    __init__.py  
    foo.py  
    bar.py  
  utils/  
    __init__.py  
    spam.py  
    grok.py  
  examples/  
    helloworld.py  
  ...
```

Чтобы превратить пакет в нечто пригодное для распространения, сначала напишите файл `setup.py`, который выглядит так:

```
# setup.py
from distutils.core import setup

setup(name='projectname',
      version='1.0',
      author='Your Name',
      author_email='you@youraddress.com',
      url='http://www.you.com/projectname',
      packages=['projectname', 'projectname.utils'],
)
```

Далее создайте файл `MANIFEST.in`, в котором перечислите файлы, которые не содержат исходный код, но которые вы хотите включить в ваш пакет:

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

Убедитесь, что `setup.py` и `MANIFEST.in` лежат в каталоге высшего уровня вашего пакета. Когда вы их туда поместите, то сможете создать исходник для распространения, напечатав такую команду:

```
% bash python3 setup.py sdist
```

Это создаст файл `projectname-1.0.zip` или `projectname-1.0.tar.gz` (зависит от платформы). Если всё это сработает, вы получите файл, который можно передавать другим людям или загружать в [Python Package Index](#).

Обсуждение

Если у вас чистый код Python, написание `setup.py` вполне бесхитростно. Потенциальный подводный камень тут в том, что вы должны вручную перечислить каждый из подкаталогов, которые составляют пакет. Обычная ошибка тут — указание только каталога высшего уровня вашего пакета, а не всех подкомпонентов. Вот почему определение `packages` в `setup.py` производится с помощью списка: `packages=['projectname', 'project name.utils']`.

Большинство программистов Python знают, что есть много сторонних инструментов для создания пакетов, такие как `setuptools`, `distribute` и т.д.

Некоторые из них заменяют библиотеку *distutils* из стандартной библиотеки. Обратите внимание, что если вы опираетесь на эти пакеты, пользователи могут столкнуться с невозможностью установить ваше ПО до тех пор, пока не установят себе требуемый менеджер пакетов. Поэтому вы не ошибётесь, если выберете самое простое решение. Как минимум, убедитесь, что ваш код может быть установлен с помощью стандартной поставки Python 3. Дополнительные возможности можно поддерживать в качестве опции, которая пригодится тем, кому доступны дополнительные пакеты.

Упаковка и распространение кода, включающего написанные на С расширения, может быть заметно более сложным делом. В главе 15 мы коснёмся этих вопросов. В частности, обратите внимание на рецепт 15.2.*

11. Сети и веб-программирование

Эта глава посвящена вопросам использования Python в сетевых и распределённых приложениях. Здесь мы обсудим темы использования Python в качестве клиента для доступа к существующим сервисам, а также использование Python для разработки сетевых сервисов (например, сервера). Также будут показаны распространённые приёмы написания кода для совместной работы интерпретаторов и их коммуникации друг с другом.

11.1. Взаимодействие с HTTP-сервисами в роли клиента

Задача

Вам нужно обращаться к различным веб-сервисам через HTTP в роли клиента. Например, скачивать данные или взаимодействовать с REST API.

Решение

Для простых задач обычно достаточно модуля *urllib.request*. Например, чтобы послать простой GET-запрос удалённому сервису, сделайте так:

```
from urllib import request, parse

# Базовый URL, к которому обращаемся
url = 'http://httpbin.org/get'

# Словарь параметров запроса (если они есть)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Кодируем строку запроса
querystring = parse.urlencode(parms)

# Делаем GET-запрос и читаем ответ
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

Если вам нужно послать параметры в теле запроса, используя метод POST, закодируйте их и предоставьте в качестве необязательных аргументов в *urlopen()*:

```
from urllib import request, parse

# Базовый URL, к которому обращаемся
url = 'http://httpbin.org/post'

# Словарь параметров запроса (если они есть)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Кодируем строку запроса
querystring = parse.urlencode(parms)

# Делаем POST-запрос и читаем ответ
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

Если вам нужно предоставить какие-то кастомные HTTP-заголовки в исходящем запросе (например, изменённое поле User-Agent), нужно создать содержащий их значения словарь, а также экземпляр *Request* — и передать его в *urlopen()*:

```
from urllib import request, parse
...
```

```
# Дополнительные заголовки
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii')), headers=headers)

# Делаем запрос и читаем ответ
u = request.urlopen(req)
resp = u.read()
```

Если ваше взаимодействие с сервисом сложнее, вам стоит обратить внимание на библиотеку [requests](#). Например, вот эквивалентный предыдущим операциям код на *requests*:

```
import requests

# Базовый URL, к которому обращаемся
url = 'http://httpbin.org/post'

# Словарь параметров запроса (если они есть)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Дополнительные заголовки
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Раскодированный текст, возвращённый запросом
text = resp.text
```

Стоит отметить, как *requests* возвращает полученное содержимое ответа на запрос. Как показано выше, атрибут *resp.text* предоставляет декодированный в Unicode текст запроса. Однако если вы обратитесь к *resp.content*, то получите не текст, а сырое бинарное содержимое. С другой стороны, если вы обратитесь к *resp.json*, то вы получите содержимое ответа в формате JSON.

Вот пример использования *requests* для создания запроса HEAD и

извлечения нескольких полей данных заголовка из ответа:

```
import requests

resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']
```

Вот пример, который показывает, как можно залогиниться в Python Package Index, используя базовую аутентификацию:

```
import requests
resp = requests.get('http://pypi.python.org/pypi?:action=login',
                    auth=('user', 'password'))
```

Вот пример передачи HTTP-куки из одного запроса следующему:

```
import requests

# Первый запрос
resp1 = requests.get(url)
...

# Второй запрос с cookie, полученными при первом запросе
resp2 = requests.get(url, cookies=resp1.cookies)
```

Последний пример — загрузка данных на сервер:

```
import requests
url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

Обсуждение

Для простого клиентского HTTP-кода встроенного модуля *urllib* обычно хватает. Однако если вам нужно что-то помимо простых POST- и GET-запросов, вы не можете полагаться на его функциональность. В этом случае лучше подойдут сторонние модули, такие как *requests*.

Например, если вы решите полностью опираться только на стандартную библиотеку вместо библиотеки типа *requests*, вам придётся реализовать ваш код с использованием низкоуровневого модуля *http.client*. Например, этот код выполняет HEAD-запрос:

```
from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)
```

А если вам нужно написать код, использующий прокси, аутентификацию, куки и другие подобные моменты, использовать *urllib* неудобно, а код получится многословным. Например, вот пример кода, который производит авторизацию в Python Package Index:

```
import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi', 'http://pypi.python.org', 'username', 'password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?action=login')
u = opener.open(r)
resp = u.read()

# Далее вы можете обращаться к другим страницам, используя opener
...
```

Если честно, всё это проще сделать с помощью *requests*.

Тестирование клиентского HTTP-кода в процессе разработки часто может быть непростым из-за большого количества неочевидных деталей, о которых нужно думать (например, куки, авторизация, заголовки, кодировки и т.п.) Если вам нужно тестировать код, посмотрите на сервис [httpbin](http://httpbin.org). Этот сайт получает запросы и «эхом» отсылает информацию обратно в форме JSON-ответа. Вот интерактивный сеанс:

```
>>> import requests
```

```
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...     headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
'keep-alive', 'Host': 'httpbin.org', 'Accept': '*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

Работать с сайтом типа [httpbin](http://httpbin.org) часто предпочтительнее, чем экспериментировать с реальным сайтом — особенно в том случае, если существует риск отключения аккаунта после трёх провалившихся попыток залогиниться (не пробуйте учиться писать клиент с HTTP-аутентификацией путём отправки запросов на сайт вашего банка).

Хотя мы здесь это и не обсудили, *requests* поддерживает намного более продвинутые клиентские протоколы HTTP, такие как OAuth. Рекомендуем вам обратиться к [замечательной документации requests](#), которая раскрывает тему намного лучше, чем наш короткий экскурс.

11.2. Создание TCP-сервера

Задача

Вы хотите реализовать сервер, который общается с клиентами по протоколу TCP.

Решение

Есть простой способ создать TCP-сервер: использовать библиотеку *socketserver*. Например, вот простой эхо-сервер:

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
```

```
    self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

Здесь вы определяете специальный класс-обработчик, который реализует метод *handle()* для обслуживания соединений с клиентами. Атрибут *request* – это клиентский сокет, а *client_address* содержит адрес клиента.

Чтобы протестировать сервер, запустите его и откройте отдельный процесс Python, который с ним соединится:

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
>>> s.recv(8192)
b'Hello'
>>>
```

Во многих случаях может быть проще определить немного другой обработчик. Вот пример, который использует базовый класс *StreamRequestHandler*, чтобы «натянуть» файлоподобный интерфейс на сокет:

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile – это файлоподобный объект для чтения
        for line in self.rfile:
            # self.wfile – это файлоподобный объект для записи
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

Обсуждение

socketserver делает создание простых TCP-серверов относительно простым. Однако вам стоит знать, что по умолчанию эти серверы являются однопоточными и могут обслуживать только одного клиента одновременно.

Если вы хотите обрабатывать множество клиентов, создайте либо экземпляр *ForkingTCPServer*, либо *ThreadingTCPServer*. Например:

```
from socketserver import ThreadingTCPServer  
...  
  
if __name__ == '__main__':  
    serv = ThreadingTCPServer(('', 20000), EchoHandler)  
    serv.serve_forever()
```

Проблема с создающими новые процессы или потоки серверами в том, что они создают новый процесс или поток на каждое соединение с клиентом. Поскольку ограничения на разрешенное количество соединений нет, злонамеренный хакер может запустить большое количество одновременных соединений и вызвать сбои в работе вашего сервера.

Если это в вашем случае это может произойти, вы можете создать выделенный заранее пул рабочих потоков или процессов. Чтобы сделать это, вы создаете экземпляр обычного (не потокового) сервера, но затем запускаете метод *serve_forever()* в пуле множества потоков. Например:

```
...  
if __name__ == '__main__':  
    from threading import Thread  
    NWORKERS = 16  
    serv = TCPServer(('', 20000), EchoHandler)  
    for n in range(NWORKERS):  
        t = Thread(target=serv.serve_forever)  
        t.daemon = True  
        t.start()  
    serv.serve_forever()
```

Обычно *TCPServer* связывается с сокетом и активирует его во время создания экземпляра. Однако иногда вы можете пожелать настроить сокет путём передачи параметров. Чтобы это сделать, предоставьте аргумент *bind_and_activate=False*:

```
if __name__ == '__main__':  
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=False)  
    # Установить различные параметры сокета  
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)  
    # Связать и активировать  
    serv.server_bind()  
    serv.server_activate()  
    serv.serve_forever()
```

Показанная выше опция настройки сокета очень распространена: она позволяет серверу перепривязаться к ранее использованному номеру порта. Это настолько стандартная штука, что стала переменной класса, которая может быть настроена в *TCPServer*. Установите её на нужное значение перед созданием экземпляра сервера, как показано в этом примере:

```
...
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()
```

В этом решении показаны два разных базовых класса-обработчика (*BaseRequestHandler* и *StreamRequestHandler*). Класс *StreamRequestHandler* на самом деле является более гибким и поддерживает некоторые возможности, которые могут быть включены через указание дополнительных переменных класса. Например:

```
import socket

class EchoHandler(StreamRequestHandler):
    # Необязательные установки (показаны дефолтные)
    timeout = 5          # Таймаут на все операции с сокетами
    rbufsize = -1         # Размер буфера чтения
    wbufsize = 0           # Размер буфера записи
    disable_nagle_algorithm = False # Устанавливает опцию TCP_NODELAY
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile – файлоподобный объект для чтения
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')
```

И, наконец, стоит отметить, что большинство высокоуровневых сетевых модулей Python (HTTP, XML-RPC и т.п.) построены на основе функциональности *socketserver*. Тем не менее, несложно реализовывать серверы напрямую, используя библиотеку *socket*. Вот простой пример прямого написания сервера с помощью сокетов:

```
from socket import socket, AF_INET, SOCK_STREAM
```

```
def echo_handler(address, client_sock):
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server('', 20000)
```

11.3. Создание UDP-сервера

Задача

Вы хотите реализовать сервер, который общается с клиентами по протоколу UDP.

Решение

Как и в случае с TCP, UDP-серверы легко создать с помощью библиотеки `socketserver`. Например, вот простой сервер, отвечающий на запрос текущим временем:

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Получаем сообщение и клиентский сокет
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
```

```
serv = UDPServer( '', 20000 ), TimeHandler)
serv.serve_forever()
```

Как и ранее, вы определяете специальный класс-обработчик, в котором реализован метод *handle()*, предназначенный для обслуживания соединений с клиентами. Атрибут *request* — это кортеж, который содержит входящую датаграмму и объект сокета для сервера. *client_address* содержит адрес клиента.

Чтобы протестировать сервер, запустите его и откройте отдельный процесс Python, который будет посыпать ему сообщения:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

Обсуждение

Типичный UDP-сервер получает входящую датаграмму (сообщение) вместе с адресом клиента. Если присутствует отвечающий сервер, он посыпает датаграмму обратно клиенту. Для передачи датаграмм вы должны использовать методы сокета *sendto()* и *recvfrom()*. Хотя традиционные методы *send()* и *recv()* также могут работать, два предыдущих чаще используются в UDP-соединениях.

Поскольку протокол не подразумевает установку соединения, UDP-серверы чаще оказываются проще в написании, нежели TCP-серверы. Однако UDP имеет врождённый недостаток — ненадёжность («соединение» не устанавливается, и сообщения могут быть потеряны). Поэтому именно вам придется решать, как разбираться с потерянными сообщениями. Эта тема лежит за пределами данной книги, но обычно вам потребуется добавить в программу последовательные номера, повторы, таймауты и другие механизмы подтверждения надёжности, если она важна для вашего приложения. UDP часто используется в случаях, когда отсутствуют строгие требования к надёжности доставки. Например, в приложениях реального времени, таких как сервисы стриминга мультимедийного контента и игры, просто отсутствует возможность отправиться назад во времени и

восстановить потерянный пакет, так что программа просто пропускает его и продолжает работу.

Класс *UDPServer* является однопоточным, что означает возможность обслуживания только одного запроса одновременно. На практике для UDP-соединений это менее важно, чем для TCP. Однако если вы хотите конкурентной работы, создавайте экземпляры *ForkingUDPServer* или *ThreadingUDPServer*:

```
from socketserver import ThreadingUDPServer
...
if __name__ == '__main__':
    serv = ThreadingUDPServer(('',20000), TimeHandler)
    serv.serve_forever()
```

Реализовать UDP-сервер напрямую, через сокеты, тоже несложно. Вот пример:

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), addr)

if __name__ == '__main__':
    time_server(('', 20000))
```

11.4. Генерация диапазона IP-адресов из CIDR-адреса

Задача

У вас есть сетевой CIDR-адрес типа 123.45.67.89/27, и вы хотите сгенерировать диапазон всех IP-адресов, которые он представляет (то есть 123.45.67.64, 123.45.67.65, ..., 123.45.67.95).

Решение

Для выполнения таких вычислений удобно использовать модуль *ipaddress*.

Например:

```
>>> import ipaddress
>>> net = ipaddress.ip_network('123.45.67.64/27')
>>> net
IPv4Network('123.45.67.64/27')
>>> for a in net:
...     print(a)
...
123.45.67.64
123.45.67.65
123.45.67.66
123.45.67.67
123.45.67.68
...
123.45.67.95
>>>

>>> net6 = ipaddress.ip_network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> net6
IPv6Network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> for a in net6:
...
print(a)
...
12:3456:78:90ab:cd:ef01:23:30
12:3456:78:90ab:cd:ef01:23:31
12:3456:78:90ab:cd:ef01:23:32
12:3456:78:90ab:cd:ef01:23:33
12:3456:78:90ab:cd:ef01:23:34
12:3456:78:90ab:cd:ef01:23:35
12:3456:78:90ab:cd:ef01:23:36
12:3456:78:90ab:cd:ef01:23:37
>>>
```

Сетевые объекты также позволяют индексирование по принципу массивов.

Например:

```
>>> net.num_addresses
32
>>> net[0]
IPv4Address('123.45.67.64')
>>> net[1]
IPv4Address('123.45.67.65')
>>> net[-1]
```

```
IPv4Address('123.45.67.95')
>>> net[-2]
IPv4Address('123.45.67.94')
>>>
```

Также вы можете выполнять такие операции, как проверку на принадлежность к сети:

```
>>> a = ipaddress.ip_address('123.45.67.69')
>>> a in net
True
>>> b = ipaddress.ip_address('123.45.67.123')
>>> b in net
False
>>>
```

IP-адрес и сетевой адрес могут быть определены вместе как IP-интерфейс. Например:

```
>>> inet = ipaddress.ip_interface('123.45.67.73/27')
>>> inet.network
IPv4Network('123.45.67.64/27')
>>> inet.ip
IPv4Address('123.45.67.73')
>>>
```

Обсуждение

Модуль *ipaddress* имеет классы для представления IP-адресов, сетей и интерфейсов. Это особенно полезно, если вы хотите написать код, который должен как-то манипулировать сетевыми адресами (парсинг, вывод, валидация и т.п.)

Обратите внимание, что между модулем *ipaddress* и другими связанными с сетью модулями, такими как библиотека *socket*, есть лишь ограниченная связь. В частности, обычно невозможно использовать экземпляр *IPv4Address* в качестве замены строки с адресом. Вместо этого вы должны явно конвертировать его с помощью *str()*. Например:

```
>>> a = ipaddress.ip_address('127.0.0.1')
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect((a, 8080))
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'IPv4Address' object to str implicitly
>>> s.connect((str(a), 8080))
>>>
```

См. «[Введение в модуль `ipaddress`](#)», чтобы получить больше информации и продвинутых примеров использования.

11.5. Создание простого REST-интерфейса

Задача

Вы хотите иметь возможность контролировать вашу программу или общаться с ней удалённо, по сети, используя простой REST-интерфейс. Однако вы не хотите делать это путём установки полноценного веб-фреймворка.

Решение

Один из самых простых способов построения REST-интерфейсов — это создание небольшой библиотеки на основе стандарта WSGI, как описано в [PEP 3333](#). Вот пример:

```
# resty.py

import cgi

def notfound_404(environ, start_response):
    start_response('404 Not Found', [ ('Content-type', 'text/plain') ])
    return [b'Not Found']

class PathDispatcher:
    def __init__(self):
        self.pathmap = { }

    def __call__(self, environ, start_response):
        path = environ['PATH_INFO']
        params = cgi.FieldStorage(environ['wsgi.input'],
                                  environ=environ)
        method = environ['REQUEST_METHOD'].lower()
        environ['params'] = { key: params.getvalue(key) for key in params }
        handler = self.pathmap.get((method, path), notfound_404)
        return handler(environ, start_response)
```

```
def register(self, method, path, function):
    self.pathmap[method.lower(), path] = function
    return function
```

Чтобы использовать этот диспетчер, вы просто пишете разные обработчики:

```
import time
_hello_resp = '''\
<html>
    <head>
        <title>Hello {name}</title>
    </head>
    <body>
        <h1>Hello {name}!</h1>
    </body>
</html>'''

def hello_world(environ, start_response):
    start_response('200 OK', [ ('Content-type', 'text/html')])
    params = environ['params']
    resp = _hello_resp.format(name=params.get('name'))
    yield resp.encode('utf-8')

_localtime_resp = '''\
<?xml version="1.0"?>
<time>
    <year>{t.tm_year}</year>
    <month>{t.tm_mon}</month>
    <day>{t.tm_mday}</day>
    <hour>{t.tm_hour}</hour>
    <minute>{t.tm_min}</minute>
    <second>{t.tm_sec}</second>
</time>'''

def localtime(environ, start_response):
    start_response('200 OK', [ ('Content-type', 'application/xml') ])
    resp = _localtime_resp.format(t=time.localtime())
    yield resp.encode('utf-8')

if __name__ == '__main__':
    from resty import PathDispatcher
    from wsgiref.simple_server import make_server

    # Создаем диспетчер и регистрируем функции
    dispatcher = PathDispatcher()
    dispatcher.register('GET', '/hello', hello_world)
    dispatcher.register('GET', '/localtime', localtime)

    # Запускаем базовый сервер
```

```
httpd = make_server('', 8080, dispatcher)
print('Serving on port 8080...')
httpd.serve_forever()
```

Чтобы протестировать этот сервер, вы можете обратиться к нему через браузер или *urllib*. Например:

```
>>> u = urlopen('http://localhost:8080/hello?name=Guido')
>>> print(u.read().decode('utf-8'))
<html>
  <head>
    <title>Hello Guido</title>
  </head>
  <body>
    <h1>Hello Guido!</h1>
  </body>
</html>
>>> u = urlopen('http://localhost:8080/localtime')
>>> print(u.read().decode('utf-8'))
<?xml version="1.0"?>
<time>
  <year>2012</year>
  <month>11</month>
  <day>24</day>
  <hour>14</hour>
  <minute>49</minute>
  <second>17</second>
</time>
>>>
```

Обсуждение

REST-интерфейсы обычно применяются в программах, которые должны отвечать на обычные HTTP-запросы. Однако, в отличие от полноценного сайта, они часто просто передают данные. Эти данные могут быть закодированы в различных форматах, таких как XML, JSON или CSV. Хотя это выглядит минималистично, предоставление такого API может быть крайне полезной штукой для самых разных применений.

Например, программы, которые работают долго, могут использовать REST API для реализации мониторинга или диагностики. Приложения для работы с big data могут использовать REST для построения систем извлечения данных по запросу. REST может быть даже использован для управления устройствами, такими как роботы, сенсоры, кофемолки или лампочки. Более того, REST API отлично поддерживается различными программными

окружениями для клиентской части, такими как JavaScript, Android, iOS и т.п. Предоставление такого интерфейса может быть отличным способом вдохновить разработку более сложных приложений, взаимодействующих с вашим кодом через REST.

Чтобы реализовать простой REST-интерфейс, часто достаточно просто написать код на основе стандарта Python WSGI. Он поддерживается стандартной библиотекой, но также большинством сторонних фреймворков. Так что если вы используете его, то ваш код становится гибче и расширяет возможности дальнейшего использования.

Согласно WSGI, вы просто реализуете приложения в форме вызываемых объектов с такими условиями вызова:

```
import cgi

def wsgi_app(environ, start_response):
    ...
```

Аргумент `environ` — это словарь, который содержит значения, которые произошли от интерфейса CGI, который предоставляется различными веб-серверами, такими как Apache (см. [Internet RFC 3875](#)). Чтобы извлекать различные поля, вы должны написать такой код:

```
def wsgi_app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    # Парсим параметры запроса
    params = cgi.FieldStorage(environ['wsgi.input'], environ=environ)
    ...
```

Здесь показано несколько распространённых значений.
`environ['REQUEST_METHOD']` — это тип запроса (т.е., GET, POST, HEAD и т.п.)
`environ['PATH_INFO']` — это запрашиваемый путь или ресурс. Вызов `cgi.FieldStorage()` извлекает предоставленные параметры запроса из запроса и помещает их в словареподобный объект для дальнейшего использования.

Аргумент `start_response` — это функция, которая должна быть вызвана, чтобы инициировать ответ. Первый аргумент — это получившийся HTTP-статус. Второй аргумент — это список кортежей (`name, value`), которые составляют HTTP-заголовки ответа. Например:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
```

Чтобы вернуть данные, WSGI-приложение должно вернуть последовательность байтовых строк. Это может быть сделано с использованием списка:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    resp = []
    resp.append(b'Hello World\n')
    resp.append(b'Goodbye!\n')
    return resp
```

Или же вы можете использовать *yield*:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    yield b'Hello World\n'
    yield b'Goodbye!\n'
```

Важно подчеркнуть, что эти байтовые строки должны быть использованы в результате. Если ответ состоит из текста, его нужно будет сначала закодировать в байты. Конечно, нет такого требования, чтобы возвращаемое значение было текстом — вы можете легко написать приложение, которое создает картинки.

Хотя WSGI-приложения часто определяются как функция, но может быть использован и экземпляр, если в нём реализован подходящий метод `__call__()`. Например:

```
class WSGIApplication:
    def __init__(self):
        ...
    def __call__(self, environ, start_response):
        ...
```

Этот приём был использован для создания класса *PathDispatcher* в этом рецепте. Диспетчер ничего не делает, кроме как управляет отображением пар словаря (*method, path*) в функции-обработчики. Когда приходит запрос,

метод и путь извлекаются и используются для диспетчериизации на обработчик. Также любые переменные запроса парсятся и помещаются в словарь, который сохраняется как `environ['params']` (этот последний шаг настолько распространён, что имеет смысл делать это в диспетчере, чтобы избежать дублирования кода).

Чтобы использовать диспетчер, вы просто создаете экземпляр и регистрируете различные WSGI-функции приложения с его помощью, как показано в рецепте. Написание этих функций обычно абсолютно бесхитростно, если вы следуете правилам, касающимся функции `start_response()` и производите вывод в форме байтовых строк.

Момент, о котором стоит помнить при написании этих функций, касается осторожного подхода к использованию строковых шаблонов. Никто не любит работать с кодом, который представляет собой перепутанную массу функций `print()`, XML и различных операций форматирования. В решении определены и используются строковые шаблоны в тройных кавычках. Этот подход облегчает изменение формата вывода позже (просто измените шаблон, а использующий его код менять не придётся).

Наконец, важный момент использования WSGI заключается в том, что ничто в этой реализации не специфично для конкретного веб-сервера. Это и есть главная идея — поскольку стандарт нейтрален по отношению к серверам и фреймворкам, вы сможете прикрутить ваше приложение к практически любому серверу. В рецепте для проверки используется такой код:

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    # Создаём диспетчер и регистрируем функции
    dispatcher = PathDispatcher()
    ...

    # Запускаем базовый сервер
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()
```

Это создаёт простой сервер, который вы можете использовать, чтобы проверить, работает ли ваша реализация. Позже, когда вы будете готовы к масштабированию, вы измените этот код, чтобы он работал с конкретным сервером.

WSGI – это спецификация, которая намеренно сделана минималистичной. Поэтому она не предоставляет поддержку более продвинутых концепций, таких как аутентификация, куки, редиректы и так далее. Все они легко реализуются самостоятельно. Однако если вам нужно чуть-чуть больше функций, вы можете посмотреть на сторонние библиотеки типа [WebOb](#) или [Paste](#).

11.6. Реализация простого удалённого вызова процедуры через XML-RPC

Задача

Вам нужен простой способ выполнить функции или методы в программах Python, работающих на удалённых компьютерах.

Решение

Вероятно, самый лёгкий способ реализовать простой механизм вызова удалённой процедуры – это XML-RPC. Вот пример простого сервера, который реализует простое хранилище типа «ключ-значение» (key-value store):

```
from xmlrpc.server import SimpleXMLRPCServer

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, address):
        self._data = {}
        self._serv = SimpleXMLRPCServer(address, allow_none=True)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data
```

```

def keys(self):
    return list(self._data)

def serve_forever(self):
    self._serv.serve_forever()

# Пример
if __name__ == '__main__':
    kvserv = KeyValueServer(('', 15000))
    kvserv.serve_forever()

```

Вот как вы можете обращаться к серверу удалённо из клиента:

```

>>> from xmlrpclib import ServerProxy
>>> s = ServerProxy('http://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>

```

Обсуждение

XML-RPC может быть чрезвычайно простым способом реализовать сервис для удалённого вызова процедур. Вам нужно только создать экземпляр сервера, зарегистрировать функции с использованием метода `register_functions()` и запустить его с помощью метода `serve_forever()`. Данный рецепт упаковывает всё это в класс, но такого требования на самом деле нет. Например, вы можете создать сервер самостоятельно:

```

from xmlrpclib import SimpleXMLRPCServer
def add(x,y):
    return x+y

serv = SimpleXMLRPCServer(('', 15000))
serv.register_function(add)
serv.serve_forever()

```

Функции, которые показываются наружу через XML-RPC, работают только с некоторыми типами данных, такими как строки, числа, списки и словари. Для всего остального нужно будет изучать тему. Например, если вы передадите экземпляр через XML-RPC, то будет обработан только словарь экземпляра:

```
>>> class Point:  
...     def __init__(self, x, y):  
...         self.x = x  
...         self.y = y  
...  
>>> p = Point(2, 3)  
>>> s.set('foo', p)  
>>> s.get('foo')  
{'x': 2, 'y': 3}  
>>>
```

Обработка бинарных данных также немного отличается от того, что вы могли бы ожидать:

```
>>> s.set('foo', b'Hello World')  
>>> s.get('foo')  
<xmlrpc.client.Binary object at 0x10131d410>  
>>> _.data  
b'Hello World'  
>>>
```

В качестве общего правила: наверное, не стоит показывать XML-RPC-сервис всему миру в качестве публичного API. Этот подход хорошо работает только для внутренних сетей, где вы можете писать простые распределённые программы, работающие на нескольких компьютерах.

Недостаток XML-RPC — это его производительность. Реализация *SimpleXMLRPCServer* является однопоточной, что неприемлемо для крупных приложений (хотя и его можно запускать многопоточно — см. [рецепт 11.2](#)). Также, поскольку XML-RPC сериализует все данные в XML, он будет по определению медленнее других подходов. Однако преимущество этой кодировки состоит в том, что её понимают очень многие языки программирования. Поэтому клиенты, написанные не на Python, смогут обращаться к вашему сервису.

Несмотря на свои ограничения, о XML-RPC стоит знать — на случай, если вам понадобится быстро и начерно сделать систему удалённого вызова

процедур. Часто этого простого решения вполне достаточно.

11.7. Простое взаимодействие между интерпретаторами

Задача

Вы запускаете множество копий интерпретатора Python — возможно, на разных компьютерах, — и вы хотели бы обмениваться данными между интерпретаторами через сообщения.

Решение

С помощью модуля *multiprocessing.connection* достаточно легко наладить общение между интерпретаторами. Вот простой пример эхо-сервера:

```
from multiprocessing.connection import Listener
import traceback

def echo_client(conn):
    try:
        while True:
            msg = conn.recv()
            conn.send(msg)
    except EOFError:
        print('Connection closed')

def echo_server(address, authkey):
    serv = Listener(address, authkey=authkey)
    while True:
        try:
            client = serv.accept()
            echo_client(client)
        except Exception:
            traceback.print_exc()

echo_server(('', 25000), authkey=b'peekaboo')
```

Вот простой пример соединения клиента с сервером и отправки сообщений:

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 25000), authkey=b'peekaboo')
>>> c.send('hello')
```

```
>>> c.recv()
'hello'
>>> c.send(42)
>>> c.recv()
42
>>> c.send([1, 2, 3, 4, 5])
>>> c.recv()
[1, 2, 3, 4, 5]
>>>
```

В отличие от низкоуровневого сокета, сообщения остаются неизменными (каждый объект, отосланный с использованием `send()`, получается целиком и полностью с помощью `recv()`). Объекты сериализованы с помощью `pickle`. Так что любой совместимый с `pickle` объект может быть отправлен или получен через это соединение.

Обсуждение

Есть много пакетов и библиотек, которые связаны с той или иной формой передачи сообщений: ZeroMQ, Celery и т.д. В качестве альтернативы вы можете склониться к написанию слоя сообщений над низкоуровневыми сокетами. Однако иногда вам нужно простое решение. Библиотека `multiprocessing.connection` именно такова: используя несколько простых примитивов, вы легко соедините интерпретаторы вместе и заставите их обмениваться сообщениями.

Если вы знаете, что интерпретаторы будут запускаться на одном и том же компьютере, вы можете использовать альтернативную форму сетевых взаимодействий, такую как сокеты домена UNIX или именованные каналы Windows. Чтобы создать соединение через сокет домена UNIX, просто измените адрес на имя файла:

```
s = Listener('/tmp/myconn', authkey=b'peekaboo')
```

Чтобы создать соединение через именованный канал Windows, используйте имя файла таким образом:

```
s = Listener(r'\\.\pipe\myconn', authkey=b'peekaboo')
```

Общее правило: вы не должны использовать `multiprocessing` для реализации публичных сервисов. Параметр `authkey`, передаваемый `Client()` и `Listener()`, помогает аутентифицировать конечные точки соединения. Попытки

соединения с неправильным ключом будут возбуждать исключение. Этот модуль лучше подходит для длительных соединений (но не для большого количества коротких соединений). Например, два интерпретатора могут установить соединение при запуске и поддерживать соединение активным на всём протяжении выполнения задачи.

Не используйте *multiprocessing*, если вам нужен более низкоуровневый контроль над аспектами соединения. Например, если вы хотите использовать таймауты, неблокирующий ввод-вывод или что-то похожее, вам стоит использовать другую библиотеку или реализовать эти возможности поверх сокетов.

11.8. Реализация вызовов удалённых процедур

Задача

Вы хотите реализовать простую систему удаленного вызова процедур (RPC) поверх слоя передачи сообщений, такого как сокеты, соединения модуля *multiprocessing* или ZeroMQ.

Решение

RPC легко реализовать путём упаковки с помощью *pickle* запросов, аргументов и возвращаемых значений функций, и передачи упакованной байтовой строки между интерпретаторами. Вот пример простого RPC-обработчика, который можно встроить в сервер:

```
# rpcserver.py

import pickle

class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
```

```

while True:
    # Получаем сообщение
    func_name, args, kwargs = pickle.loads(connection.recv())
    # Запускаем RPC и посылаем ответ
    try:
        r = self._functions[func_name](*args, **kwargs)
        connection.send(pickle.dumps(r))
    except Exception as e:
        connection.send(pickle.dumps(e))
    except EOFError:
        pass

```

Чтобы использовать этот обработчик, вам нужно добавить его в сервер сообщений. Существует множество возможных выборов, но библиотека *multiprocessing* — один из самых простых вариантов. Вот пример RPC-сервера:

```

from multiprocessing.connection import Listener
from threading import Thread

def rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client,))
        t.daemon = True
        t.start()

# Удалённые функции
def add(x, y):
    return x + y

def sub(x, y):
    return x - y

# Регистрируем с обработчиком
handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

# Запускаем сервер
rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')

```

Чтобы обратиться к серверу с удалённого клиента, вам нужно создать соответствующий RPC-прокси-класс, который будет перенаправлять запросы. Например:

```
import pickle

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection

    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwargs)))
            result = pickle.loads(self._connection.recv())
            if isinstance(result, Exception):
                raise result
            return result
        return do_rpc
```

Чтобы использовать прокси, оберните его вокруг соединения с сервером.

Например:

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)
5
>>> proxy.sub(2, 3)
-1
>>> proxy.sub([1, 2], 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "rpcserver.py", line 37, in do_rpc
    raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

Стоит отметить, что многие слои передачи сообщений (такие как *multiprocessing*) уже сериализуют данные с помощью *pickle*. В этом случае вызовы *pickle.dumps()* и *pickle.loads()* могут быть удалены.

Обсуждение

Главная идея классов *RPCHandler* и *RPCProxy* относительно проста. Если клиент хочет вызывать удалённую функцию, такую как *foo(1, 2, z=3)*, прокси-класс создает кортеж ('*foo*', (1, 2), {'z': 3}), который содержит имя функции и аргументы. Этот кортеж упаковывается и отсылается через соединение. Это выполняется в замыкании *do_rpc()*, которое возвращается методом *__getattr__()* класса *RPCProxy*. Сервер получает и распаковывает сообщение,

проверяет, зарегистрировано ли имя функции, а затем выполняет её с переданными аргументами. Результат (или исключение) упаковывается и отсылается обратно.

Как показано выше, данный пример использует для коммуникации модуль *multiprocessing*. Однако этот подход может быть применён для практически любой системы передачи сообщений. Например, если вы захотите реализовать RPC через ZeroMQ, просто замените объекты соединений подходящими объектами сокетов ZeroMQ.

Поскольку здесь применяется *pickle*, безопасность под угрозой (умный хакер может создать сообщения, которые выполняют произвольные функции во время распаковки). В частности, вы никогда не должны разрешать RPC от недоверенных или неавтентифицированных клиентов. И вы ни в коем случае не должны давать доступ с любого компьютера, подключенного к интернету: RPC нужно использовать внутри сети, за файерволом.

В качестве альтернативы *pickle* вы можете попробовать JSON, XML или какой-то другой способ кодирования данных для сериализации. Например, этот рецепт легко адаптировать к JSON — просто замените *pickle.loads()* и *pickle.dumps()* на *json.loads()* и *json.dumps()*. Например:

```
# jsonrpcserver.py
import json

class RPCHandler:
    def __init__(self):
        self._functions = {}

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Получаем сообщение
                func_name, args, kwargs = json.loads(connection.recv())
                # Запускаем RPC и посылаем ответ
                try:
                    r = self._functions[func_name](*args, **kwargs)
                    connection.send(json.dumps(r))
                except Exception as e:
                    connection.send(json.dumps(str(e)))
        except EOFError:
            pass
```

```
# jsonrpcclient.py
import json

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(json.dumps((name, args, kwargs)))
            result = json.loads(self._connection.recv())
            return result
        return do_rpc
```

Сложный момент в реализации RPC — обработка исключений. По меньшей мере, сервер не должен падать, если метод возбуждает исключение. Однако средства отправки сообщений об исключениях обратно клиенту требуют изучения. Если вы используете *pickle*, экземпляры исключений часто сериализуются и заново возбуждаются уже на клиенте. Если вы используете какой-либо другой протокол, вам, вероятно, придётся подумать об альтернативном подходе. Как минимум, вы, вероятно, захотите возвращать строку с исключением в ответе. Это подход, которому мы следовали в примере с JSON.

Ещё один пример реализации RPC вы найдете в [рецепте 11.6.](#), где обсуждаются классы *SimpleXMLRPCServer* и *ServerProxy*.

11.9. Простая аутентификация клиентов

Задача

Вы хотите реализовать простой способ аутентификации клиентов, соединяющихся с серверами в распределённой системе, но вам не нужны сложные решения типа SSL.

Решение

Простой, но при этом эффективный способ аутентификации может быть реализован путём хендшейка при соединении с использованием модуля *hmac*. Вот пример:

```
import hmac
```

```
import os

def client_authenticate(connection, secret_key):
    """
    Аутентифицирует клиент на удалённом сервере.
    connection представляет сетевое соединение.
    secret_key — это ключ, известный только клиенту и серверу.
    """

    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)

def server_authenticate(connection, secret_key):
    """
    Запрашивает аутентификацию клиента.
    """

    message = os.urandom(32)
    connection.send(message)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    response = connection.recv(len(digest))
    return hmac.compare_digest(digest, response)
```

Основная идея в том, что до установки соединения сервер предоставляет клиенту сообщение, состоящее из случайных байтов (в данном случае они генерируются `os.random()`). И клиент, и сервер вычисляют криптографический хэш этих случайных данных, используя `hmac` и секретный ключ, известный только обеим сторонам. Клиент посыпает вычисленный дайджест обратно на сервер, где они сравниваются, после чего принимается решение — принимать соединение или нет.

Сравнение получившихся дайджестов должно выполняться с помощью функции `hmac.compare_digest()`. Она написана таким образом, чтобы предотвратить атаки на основе анализа тайминга и должна быть использована вместо стандартного оператора сравнения (`==`).

Чтобы использовать эти функции, вы должны включить их в существующий сетевой код или код системы обмена сообщениями. Например, серверный код с сокетами может выглядеть как-то так:

```
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'
def echo_handler(client_sock):
    if not server_authenticate(client_sock, secret_key):
```

```

        client_sock.close()
        return
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        c,a = s.accept()
        echo_handler(c)

echo_server(('', 18000))

```

В клиенте нужен примерно такой код:

```

from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'

s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 18000))
client_authenticate(s, secret_key)
s.send(b'Hello World')
resp = s.recv(1024)
...

```

Обсуждение

Обычно аутентификация на базе *htmac* используется для внутренних систем обмена сообщениями и взаимодействия между процессами. Например, если вы пишете систему, в которой общаются между собой процессы, запущенные на кластере, вы можете использовать этот подход, чтобы быть уверенным, что соединяться между собой могут только процессы, у которых есть соответствующие права. Например, HMAC-аутентификация используется библиотекой *multiprocessing*, когда она устанавливает сообщение между подпроцессами.

Важно подчеркнуть, что аутентификация соединения — это не то же самое, что шифрование. Последующая коммуникация по аутентифицированному соединению идёт без шифрования, и поэтому может быть видима всем, кто

вклинился посередине и будет снiffeить трафик (хотя секретный ключ, известный обеим сторонам, никогда не передаётся).

Алгоритм аутентификации, используемый *htac*, базируется на криптографических функциях хэширования, таких как MD5 и SHA-1, и в деталях описан в [IETF RFC 2104](#).

11.10. Добавление SSL в сетевые сервисы

Задача

Вы хотите реализовать сетевой сервис, использующий сокеты, где серверы и клиенты аутентифицируют друг друга и шифруют передаваемые данные с помощью SSL.

Решение

Модуль *ssl* предоставляет поддержку для добавления SSL к низкоуровневым соединениям на базе сокетов. В частности, функция *ssl.wrap_socket()* принимает существующий сокет и оборачивает его слоем SSL. Вот, например, простой эхо-сервер, который предоставляет серверный сертификат подсоединяющимся клиентам:

```
from socket import socket, AF_INET, SOCK_STREAM
import ssl

KEYFILE = 'server_key.pem'
# Приватный ключ сервера
CERTFILE = 'server_cert.pem'
# Сертификат сервера (передаваемый клиенту)

def echo_client(s):
    while True:
        data = s.recv(8192)
        if data == b'':
            break
        s.send(data)
    s.close()
    print('Connection closed')

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
```

```

s.listen(1)

# Оборачивает слоем SSL, требуя клиентских сертификатов
s_ssl = ssl.wrap_socket(s,
                       keyfile=KEYFILE,
                       certfile=CERTFILE,
                       server_side=True
                      )

# Ждёт соединений
while True:
    try:
        c,a = s_ssl.accept()
        print('Got connection', c, a)
        echo_client(c)
    except Exception as e:
        print('{}: {}'.format(e.__class__.__name__, e))

echo_server(('', 20000))

```

Вот интерактивный сеанс, который показывает, как клиент соединяется с сервером. Клиент запрашивает у сервера его сертификат и проверяет его:

```

>>> from socket import socket, AF_INET, SOCK_STREAM
>>> import ssl
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s_ssl = ssl.wrap_socket(s,
...                         cert_reqs=ssl.CERT_REQUIRED,
...                         ca_certs = 'server_cert.pem')
>>> s_ssl.connect(('localhost', 20000))
>>> s_ssl.send(b'Hello World?')
12
>>> s_ssl.recv(8192)
b'Hello World?'
>>>

```

Проблема со всем этим низкоуровневым сокетным хакерством в том, что это не очень хорошо работает с существующими сетевыми сервисами, уже реализованными в стандартной библиотеке. Например, большая часть серверного кода (HTTP, XML-RPC и т.д.) базируется на библиотеке `socketserver`. Код клиентов тоже реализуется на более высоком уровне. В существующие сервисы можно добавить SSL, но для этого требуется немного другой подход.

Во-первых, в серверы SSL может быть добавлен с помощью класса-примеси (миксина):

```

import ssl

class SSLMixin:
    """
    Класс-миксин, который добавляет поддержку SSL существующим серверам,
    основанным на модуле socketserver.
    """

    def __init__(self, *args,
                 keyfile=None, certfile=None, ca_certs=None,
                 cert_reqs=ssl.NONE,
                 **kwargs):
        self._keyfile = keyfile
        self._certfile = certfile
        self._ca_certs = ca_certs
        self._cert_reqs = cert_reqs
        super().__init__(*args, **kwargs)

    def get_request(self):
        client, addr = super().get_request()
        client_ssl = ssl.wrap_socket(client,
                                     keyfile = self._keyfile,
                                     certfile = self._certfile,
                                     ca_certs = self._ca_certs,
                                     cert_reqs = self._cert_reqs,
                                     server_side = True)
        return client_ssl, addr

```

Чтобы использовать этот миксин, вы должны примешать его к другим классам сервера. Например, вот как определить XML-RPC-сервер, который работает через SSL:

```

# XML-RPC-сервер с SSL

from xmlrpc.server import SimpleXMLRPCServer

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

```

Вот XML-RPC-сервер из [рецепта 11.6.](#), немного модифицированного для работы через SSL:

```

import ssl
from xmlrpc.server import SimpleXMLRPCServer
from sslmixin import SSLMixin

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

```

```

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, *args, **kwargs):
        self._data = {}
        self._serv = SSLSimpleXMLRPCServer(*args, allow_none=True, **kwargs)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)

    def serve_forever(self):
        self._serv.serve_forever()

if __name__ == '__main__':
    KEYFILE='server_key.pem'      # Приватный ключ сервера
    CERTFILE='server_cert.pem'   # Сертификат сервера
    kvserv = KeyValueServer(('', 15000),
                           keyfile=KEYFILE,
                           certfile=CERTFILE),
    kvserv.serve_forever()

```

Чтобы использовать этот сервер, вы можете соединиться с ним с помощью обычного модуля *xmlrpc.client*. Просто напишите *https:* в URL. Например:

```

>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('https://localhost:15000', allow_none=True)
>>> s.set('foo','bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')

```

```
>>> s.exists('spam')
False
>>>
```

Сложность работы с SSL-клиентами состоит в том, что требуются дополнительные шаги, чтобы верифицировать сертификат сервера или предоставить серверу опознавательную информацию о клиенте (такую, как клиентский сертификат). К сожалению, для реализации этого нет стандартного пути, так что вопрос потребует некоторого изучения. Вот, однако, пример того, как установить безопасное XML-RPC-соединение, которое проверяет сертификат сервера:

```
from xmlrpclib import SafeTransport, ServerProxy
import ssl

class VerifyCertSafeTransport(SafeTransport):
    def __init__(self, cafile, certfile=None, keyfile=None):
        SafeTransport.__init__(self)
        self._ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
        self._ssl_context.load_verify_locations(cafile)
        if cert:
            self._ssl_context.load_cert_chain(certfile, keyfile)
        self._ssl_context.verify_mode = ssl.CERT_REQUIRED

    def make_connection(self, host):
        # Элементы в передаваемом словаре передаются как именованные
        # аргументы в конструктор http.client.HTTPSConnection().
        # Аргумент context позволяет экземпляру ssl.SSLContext
        # передаваться с информацией о конфигурации SSL.
        s = super().make_connection((host, {'context': self._ssl_context}))

        return s

# Создаем клиентский прокси
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem'),
                allow_none=True)
```

Как показано выше, сервер предоставляет сертификат клиенту, и клиент его проверяет. Эта верификация может быть двунаправленной. Если сервер хочет верифицировать клиента, измените процесс его запуска на следующий:

```
'client_key.pem') ,  
allow_none=True)
```

Обсуждение

Этот рецепт испытает на прочность ваши знания в области системного конфигурирования и SSL. Самое сложное — произвести первоначальную конфигурацию ключей, сертификатов и всех прочих необходимых компонентов по порядку.

Поясним: каждая конечная точка SSL-соединения обычно имеет приватный ключ и подписанный файл сертификата. Файл сертификата содержит публичный ключ. Он предоставляется удалённому пиру (*peer*) при каждом соединении. Сертификаты публичных серверов обычно подписываются центром выдачи сертификатов типа Verisign, Equifax или похожими организациями (это стоит денег). Чтобы верифицировать серверные сертификаты, клиенты поддерживают файл, в котором хранятся сертификаты доверенных центров сертификации. Например, веб-браузеры поддерживают сертификаты, соответствующие главным центрам выдачи сертификатов, и используют их для проверки целостности сертификатов, предоставляемых веб-серверами при HTTPS-соединениях.

Для целей этого рецепта вы можете создать самоподписанный сертификат. Вот как это делается:

```
bash % openssl req -new -x509 -days 365 -nodes -out server_cert.pem \  
-keyout server_key.pem  
Generating a 1024 bit RSA private key  
.....+++++  
...+++++  
writing new private key to 'server_key.pem'  
----  
You are about to be asked to enter information that will be incorporated  
into your certificate request.  
What you are about to enter is what is called a Distinguished Name or a  
There are quite a few fields but you can leave some blank  
For some fields there will be a default value,  
If you enter '.', the field will be left blank.  
----  
Country Name (2 letter code) [AU]:US  
State or Province Name (full name) [Some-State]:Illinois  
Locality Name (eg, city) []:Chicago  
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dabeaz, LLC  
Organizational Unit Name (eg, section) []:
```

```
Common Name (eg, YOUR name) []:localhost
```

```
Email Address []:
```

```
bash %
```

При создании сертификата значения для различных полей часто могут быть произвольными. Однако поле “Common Name” часто содержит имя хоста DNS сервера. Если вы просто экспериментируете на своём компьютере, используйте “localhost”. В противном случае используйте доменное имя компьютера, на котором будет запущен сервер.

В качестве результата этой конфигурации вы получите файл *server_key.pem*, содержащий приватный ключ. Он выглядит примерно так:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCZrCNLoEyAKF+f9UNCFAz50sa6jf7qkbU18si5xQrY3ZYC7juu
nL1dzLn/VbEFlITAUOgvBtPv1qUWTJGwga62VSG1oFE0ODIX3g2Nh4sRf+rySsx2
L4442nx0z405vJQ7k6eRNHAZUUnCL50+YvjiyLyt7ryLSjSuKhCcJsBZgPwIDAQAB
AoGAB5evrr7eyL4160tM5rHTeATlaLY3UBoe5Z8XN8Z6gLiB/ucSX9AysviVD/6F
3oD6z2aL8jbeJc1vHqjt0dC2dwwm32vV18mRdyoAsOpWmiqXrkvp4Bs104VpBeHw
Qt8xNSW9SFhceL3LEvv9M8i9MV39viih1ILyH8OuHdvJyFECQQDLEj12d2ppxND9
PoLqVFAirDfx2JnLTdWbc+M11a9Jdn3hKF8TcxEnFVs5Gav1MusicY5KB0y1YPb
YbTvqKc7AkEAwbnRBO2VYEzsJzp2X0IZqP9ovWokkpYx+PE4+c6MySDgaMcigL7v
WDIHJG1CHudD09GbqENasDzyb2HAIW4CzQJBADdkv+xoW6gJx42Auc2WzTcUHCA
eXR/+BLpPrhKykzbvOQ8YvS5W764SU01u1LWs3G+wnRMvrRvlMCZKgggBjkCQQCG
Jewto2+a+wkOKQxrNNScCDE5aPTmZQc5wACYq4UmCZQcOjkUOin3ST1U5iuxRqfb
v/yX6fw0qh+fLWtkOs/JAKA+okMSxZwqRtfgOFGBfwQ8/iKrniZeanTQ3L6scFXI
CHZxdJ3XQ6qUmNxNn7iJ7S/LDawo1QfwkCfd9FYoxBlg
-----END RSA PRIVATE KEY-----
```

Серверный сертификат в файле *server_cert.pem* выглядит похожим образом:

```
-----BEGIN CERTIFICATE-----
MIIC+DCCAmGgAwIBAgIJAPMd+vi45js3MA0GCSqGSIb3DQEBCQUAMFwxCzAJBgNV
BAYTA1VTMREwDwYDVQQIEwhJbGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIG
A1UEChMLRGFiZWf6LCBMTEMxEjAQBgNVBAMTCWxvY2FsaG9zdDAeFw0xMzAxMTE
ODQyMjdaFw0xNDAxMTExODQyMjdaMFwxCzAJBgNVBAYTA1VTMREwDwYDVQQIEwhJ
bGxpbm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWf6LCBMTEMx
EjAQBgNVBAMTCWxvY2FsaG9zdDCBnzANBqkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA
mawjS6BMgChfn/VDXBWs+TrGu3+6pG1JfLIucUK2N2WAu47rpy9XWS5/1WxBSC
21DoLwbT79a1FkyRsIGut1UhtaBRNDgyMd4NjYeLEX/q8krMdi+OONp8dM+DubYU
O5OnkTRwGVFJwi+dPmL48i8re68i0o0rioQnCbG2YD8CAwEAAaOBwTCBvjAdBgNV
HQ4EFgQUrtoLHHgXiDZTr26NMmgKJLJLFtIwgY4GA1UdIwSBhjCBg4AUrtoLHHgX
iDZTr26NMmgKJLJLFtKhYKRcEMFwxCzAJBgNVBAYTA1VTMREwDwYDVQQIEwhJbGxp
bm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWf6LCBMTEMxEjAQ
BgNVBAMTCWxvY2FsaG9zdIIJAPMd+vi45js3MAwGA1UdEwQFMAMBAf8wDQYJKoZI
hvcNAQEFBQADgYEAFci+dqvMG4xF8UTnbGVvZJPIzJDRee6Nb6AHQo9pOdAIMAu
WsgCp1SOaDNdKKz1+b2UT2Zp3AIW4Od51bouSNnR4M/qnr9ZD1ZctFd3iS+C5XRp
```

D3vvvcW51AnCCC80P6rXy7d7hTeFu5EYKtRGXNvVNd/06NALGDflrrOwxF3Y=

-----END CERTIFICATE-----

В серверном коде и приватный ключ, и файл сертификата будут передаваться различным связанным с SSL функциям-обёрткам. Сертификат — это то, что будет представлено клиентам. Приватный ключ должен быть защищён и оставаться на сервере.

В коде клиента нужно поддерживать специальный файл валидных сертифицирующих организаций — он нужен, чтобы верифицировать сертификат сервера. Если такого файла у вас нет, то, по крайней мере, вы должны поместить копию серверного сертификата на клиентском компьютере и использовать его как способ верификации. Во время соединения сервер представит свой сертификат, и затем вы используете сохранённый сертификат, который у вас есть, чтобы проверить корректность серверного.

Серверы также могут верифицировать клиентов. Чтобы сделать это, клиентам нужно иметь собственный приватный ключ и сертификатный ключ. Сервер также поддерживает файл с доверенными центрами сертификации, чтобы проверять клиентские сертификаты.

Если вы намереваетесь добавить поддержку SSL в реальный сетевой сервис, этот рецепт может лишь кивнуть в направлении, куда вам надлежит копать. Вам обязательно придётся свериться с [документацией](#), чтобы разобраться в вопросе. Приготовьтесь провести немало времени в попытках заставить всё заработать.

11.11. Передача файловых дескрипторов сокетов между процессами

Задача

У вас запущено несколько процессов с интерпретаторами Python, и вы хотите передать открытый файловый дескриптор из одного интерпретатора в другой. Например, у вас может быть серверный процесс, который отвечает за приём соединений, но реальное обслуживание клиентов выполняет другой интерпретатор.

Решение

Чтобы передать файловый дескриптор между процессами, вам для начала потребуется соединить процессы. На компьютерах с Unix вы можете использовать сокеты домена Unix, а на Windows — именованные каналы. Однако вместо работы с подобными низкоуровневыми механизмами, часто для установки таких соединений проще использовать модуль *multiprocessing*.

Когда соединение установлено, вы можете использовать функции *send_handle()* и *recv_handle()* из *multiprocessing.reduction*, чтобы пересыпать файловые дескрипторы между процессами. Следующий пример показывает основные моменты:

```
import multiprocessing
from multiprocessing.reduction import recv_handle, send_handle
import socket

def worker(in_p, out_p):
    out_p.close()
    while True:
        fd = recv_handle(in_p)
        print('CHILD: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd):
            while True:
                msg = s.recv(1024)
                if not msg:
                    break
                print('CHILD: RECV {!r}'.format(msg))
                s.send(msg)

def server(address, in_p, out_p, worker_pid):
    in_p.close()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(address)
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_handle(out_p, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    c1, c2 = multiprocessing.Pipe()
    worker_p = multiprocessing.Process(target=worker, args=(c1,c2))
    worker_p.start()
```

```
server_p = multiprocessing.Process(target=server,
                                    args=((' ', 15000), c1, c2, worker))
server_p.start()

c1.close()
c2.close()
```

В этом примере создаются два процесса и соединяются через объект *Pipe* из *multiprocessing*. Серверный процесс открывает сокет и ждёт соединений с клиентами. Процесс-воркер ждёт получения файлового дескриптора по каналу, используя *recv_handle()*. Когда сервер получает соединение, он посыпает получившийся файловый дескриптор сокета воркеру, используя *send_handle()*. Воркер принимает сокет и эхом отправляет данные обратно клиенту, пока соединение не закроется.

Если вы соединитесь с запущенным сервером с помощью Telnet или похожего инструмента, то вы увидите что-то подобное:

```
bash % python3 passfd.py
SERVER: Got connection from ('127.0.0.1', 55543)
CHILD: GOT FD 7
CHILD: RECV b'Hello\r\n'
CHILD: RECV b'World\r\n'
```

Самая важная часть этого примера — тот факт, что сокет клиента, принятый сервером, на самом деле обслуживается совершенно другим процессом. Сервер всего лишь передает его, закрывает и ждёт следующего соединения.

Обсуждение

Многие программисты даже не представляют себе, что можно реализовать передачу файловых дескрипторов между процессами. Однако это иногда может стать полезным инструментом для построения масштабируемых систем. Например, на многоядерном компьютере вы можете запустить несколько экземпляров интерпретатора Python и использовать передачу файловых дескрипторов для более равномерной балансировки количества клиентов, обслуживаемых каждым из интерпретаторов.

Функции *send_handle()* и *recv_handle()*, показанные в решении, работают только с многопроцессными соединениями. Вместо использования канала, вы можете соединить интерпретаторы так, как показано в [рецепте 11.7.](#), и

это будет работать до тех пор, пока вы используете сокеты домена UNIX или каналы Windows. Например, вы можете реализовать сервер и воркер как абсолютно разные программы, которые запускаются по отдельности. Вот реализация сервера:

```
# servermp.py
from multiprocessing.connection import Listener
from multiprocessing.reduction import send_handle
import socket

def server(work_address, port):
    # Ждём подсоединения воркера
    work_serv = Listener(work_address, authkey=b'peekaboo')
    worker = work_serv.accept()
    worker_pid = worker.recv()

    # Теперь запускаем TCP/IP-сервер и посылаем клиентов воркеру
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('', port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_handle(worker, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))
```

Чтобы запустить этот сервер, вы можете напечатать команду `python3 servermp.py /tmp/ servconn 15000`. Вот соответствующий код клиента:

```
# workermp.py

from multiprocessing.connection import Client
from multiprocessing.reduction import recv_handle
import os
from socket import socket, AF_INET, SOCK_STREAM

def worker(server_address):
    serv = Client(server_address, authkey=b'peekaboo')
    serv.send(os.getpid())
```

```

while True:
    fd = recv_handle(serv)
    print('WORKER: GOT FD', fd)
    with socket(AF_INET, SOCK_STREAM, fileno=fd) as client:
        while True:
            msg = client.recv(1024)
            if not msg:
                break
        print('WORKER: RECV {!r}'.format(msg))
        client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

worker(sys.argv[1])

```

Чтобы запустить воркер, напечатайте `python3 workermp.py /tmp/servconn`. Получившаяся операция будет точно такой же, как и в примере, который использует `Pipe()`.

«Под капотом» передача файлового дескриптора использует создание сокета домена UNIX и метода сокетов `sendmsg()`. Поскольку этот приём не общеизвестен, вот другая реализация сервера, которая показывает, как передать дескрипторы, используя сокеты:

```

# server.py
import socket
import struct

def send_fd(sock, fd):
    ...
    Посыпает один файловый дескриптор.
    ...
    sock.sendmsg([b'x'],
                [(socket.SOL_SOCKET, socket.SCM_RIGHTS, struct.pack('i', fd))])
    ack = sock.recv(2)
    assert ack == b'OK'

def server(work_address, port):
    # Ждёт подсоединения воркера
    work_serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    work_serv.bind(work_address)
    work_serv.listen(1)
    worker, addr = work_serv.accept()

```

```

# Теперь запускаем TCP/IP-сервер и посылаем клиентов воркеру
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
s.bind(('',port))
s.listen(1)
while True:
    client, addr = s.accept()
    print('SERVER: Got connection from', addr)
    send_fd(worker, client.fileno())
    client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

server(sys.argv[1], int(sys.argv[2]))

```

Вот реализация воркера с использованием сокетов:

```

# worker.py
import socket
import struct

def recv_fd(sock):
    """
    Получает один файловый дескриптор.
    """

    msg, anndata, flags, addr = sock.recvmsg(1,
                                              socket.CMSG_LEN(struct.calcsize('i')))

    cmsg_level, cmsg_type, cmsg_data = anndata[0]
    assert cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_FD
    sock.sendall(b'OK')
    return struct.unpack('i', cmsg_data)[0]

def worker(server_address):
    serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    serv.connect(server_address)
    while True:
        fd = recv_fd(serv)
        print('WORKER: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd):
            while True:
                msg = client.recv(1024)
                if not msg:
                    break
        print('WORKER: RECV {!r}'.format(msg))

```

```
client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

worker(sys.argv[1])
```

Если собираетесь использовать передачу файловых дескрипторов в своей программе, рекомендуем почитать более продвинутый материал, такой как *Unix Network Programming* У. Ричарда Стивенса. Передача файловых дескрипторов в Windows использует другие приёмы (не показанные здесь). Для работы с этой ОС рекомендуем внимательно изучить исходный код *multiprocessing.reduction* и понять, как он работает.

11.12. Разбираемся с вводом-выводом, управляемым событиями (event-driven I/O)

Задача

Вы слышали о пакетах, основанных на «управляемом событиями» или «асинхронном» вводе-выводе, но вы не уверены, что разобрались в том, что это значит, как это работает, и как это может повлиять на ваши программы, если вы начнете использовать такой подход.

Решение

На фундаментальном уровне управляемый событиями ввод-вывод — это приём, который принимает базовые операции ввода-вывода (то есть чтение и запись) и преобразовывает их в события, которые должны быть обработаны вашей программой. Например, когда данные приходят в сокет, это превращается в событие «получено», которое обрабатывается каким-то методом или функцией обратного вызова (коллбэком), которую вы должны предоставить для ответа на это событие. Чтобы понять, откуда копать, приведём такой пример: управляемый событиями фреймворк может начаться с базового класса, который реализует набор базовых методов обработки событий:

```

class EventHandler:
    def fileno(self):
        'Возвращает ассоциированный файловый дескриптор'
        raise NotImplemented('must implement')

    def wants_to_receive(self):
        'Возвращает True если получение разрешено'
        return False

    def handle_receive(self):
        'Выполняет операцию получения'
        pass

    def wants_to_send(self):
        'Возвращает True, если отсылка запрошена'
        return False

    def handle_send(self):
        'Отсылает исходящие данные'
        pass

```

Экземпляры этого класса могут быть подключены к циклу, который выглядит так:

```

import select

def event_loop(handlers):
    while True:
        wants_recv = [h for h in handlers if h.wants_to_receive()]
        wants_send = [h for h in handlers if h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv,
                                              wants_send, [])
        for h in can_recv:
            h.handle_receive()
        for h in can_send:
            h.handle_send()

```

Вот и всё! Секрет цикла событий — это вызов `select()`, который опрашивает файловые дескрипторы на предмет активности. Перед вызовом `select()` цикл событий просто опрашивает все обработчики, чтобы понять, какой из них хочет принимать или посыпать. Далее он предоставляет получившиеся списки в `select()`. В результате `select()` возвращает список объектов, которые уже готовы принимать или посыпать. Запускаются соответствующие методы `handle_receive()` или `handle_send()`.

Чтобы писать приложения, создаются специфические экземпляры класса

EventHandler. Например, вот два простых обработчика, которые демонстрируют работу двух сетевых UDP-сервисов:

```
import socket
import time

class UDPServer(EventHandler):
    def __init__(self, address):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(address)

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

class UDPTimeServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)

class UDPEchoServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(8192)
        self.sock.sendto(msg, addr)

if __name__ == '__main__':
    handlers = [UDPTimeServer(('', 14000)), UDPEchoServer(('', 15000))]
    event_loop(handlers)
```

Чтобы протестировать этот код, вы можете попробовать соединиться с ним из другого интерпретатора Python:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello', ('localhost', 15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
>>>
```

Реализация TCP-сервера сложнее, поскольку каждый клиент вызывает создание нового объекта-обработчика. Вот пример TCP-эхо-клиента:

```
class TCPServer(EventHandler):
    def __init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.sock.bind(address)
        self.sock.listen(1)
        self.client_handler = client_handler
        self.handler_list = handler_list

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

    def handle_receive(self):
        client, addr = self.sock.accept()
        # Добавляет клиента в список обработчиков цикла событий
        self.handler_list.append(self.client_handler(client, self.handler_list))

class TCPClient(EventHandler):
    def __init__(self, sock, handler_list):
        self.sock = sock
        self.handler_list = handler_list
        self.outgoing = bytearray()

    def fileno(self):
        return self.sock.fileno()

    def close(self):
        self.sock.close()
        # Удаляется из списка обработчиков цикла событий
        self.handler_list.remove(self)

    def wants_to_send(self):
        return True if self.outgoing else False

    def handle_send(self):
        nsent = self.sock.send(self.outgoing)
        self.outgoing = self.outgoing[nsent:]

class TCPEchoClient(TCPClient):
    def wants_to_receive(self):
        return True

    def handle_receive(self):
        data = self.sock.recv(8192)
        if not data:
            self.close()
        else:
            self.outgoing.extend(data)
```

```
if __name__ == '__main__':
    handlers = []
    handlers.append(TCPServer(('', 16000), TCPEchoClient, handlers))
    event_loop(handlers)
```

Ключевой момент примера с TCP — это добавление и удаление клиентов из списка обработчика. На каждое соединение для клиента создаётся и добавляется в список новый обработчик. Когда соединение закрывается, каждый клиент должен позаботиться о том, чтобы удалить себя из списка.

Если вы запустите эту программу и попробуете соединиться с ней с помощью Telnet или другого похожего инструмента, то вы увидите, как она эхом отправляет полученные данные обратно. Программа, по идее, должна легко работать с многочисленными клиентами.

Обсуждение

Фактически, все управляемые событиями фреймворки работают похожим образом. Реализация деталей и общая архитектура могут сильно варьироваться, но по сути всегда имеет место быть опрашивающий цикл, который проверяет сокеты на предмет активности и выполняет операции в ответ.

Потенциальное преимущество управляемого событиями ввода-вывода в том, что он позволяет работать с большим количеством одновременных соединений без использования потоков или процессов. Вызов `select()` (или эквивалентный) может быть использован для мониторинга сотен и тысяч сокетов и ответа на возникающие в них события. События обрабатываются циклом событий поочерёдно, без необходимости использования каких-либо конкурентных примитивов.

Недостаток управляемого событиями ввода-вывода в том, что он не использует настоящую конкурентность. Если любой из методов-обработчиков событий устанавливает блокировку или выполняет длительное вычисление, то всё останавливается. Также существует проблема вызова библиотечных функций, написанных не в стиле управления событиями. Всегда существует риск того, что в каком-то библиотечном вызове возникнет блокировка, которая остановит весь цикл.

Проблемы с блокированием или долгими вычислениями могут быть решены путем отсылки выполнения работы в отдельный поток или процесс. Однако

координация потоков и процессов с циклом событий — это хитрая штука. Вот пример кода, который делает это с помощью модуля `concurrent.futures`:

```
from concurrent.futures import ThreadPoolExecutor
import os

class ThreadPoolHandler(EventHandler):
    def __init__(self, nworkers):
        if os.name == 'posix':
            self.signal_done_sock, self.done_sock = socket.socketpair()
        else:
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self.signal_done_sock = socket.socket(socket.AF_INET,
                                                   socket.SOCK_STREAM)
            self.signal_done_sock.connect(server.getsockname())
            self.done_sock, _ = server.accept()
            server.close()

        self.pending = []
        self.pool = ThreadPoolExecutor(nworkers)

    def fileno(self):
        return self.done_sock.fileno()

    # Коллбэк, который выполняется после завершения потока
    def _complete(self, callback, r):
        self.pending.append((callback, r.result()))
        self.signal_done_sock.send(b'x')

    # Запускает функцию в пуле потоков
    def run(self, func, args=(), kwargs={}, *, callback):
        r = self.pool.submit(func, *args, **kwargs)
        r.add_done_callback(lambda r: self._complete(callback, r))

    def wants_to_receive(self):
        return True

    # Запускает коллбэки завершённой работы
    def handle_receive(self):
        # Вызывает все коллбэки в очереди
        for callback, result in self.pending:
            callback(result)
            self.done_sock.recv(1)
        self.pending = []
```

В этом коде метод `run()` использован для отправки работы пулу вместе с функцией обратного вызова, которая должна быть вызвана по завершению.

Затем работа отправляется в экземпляр *ThreadPoolExecutor*. Однако по-настоящему сложная проблема касается координации вычисленного результата и цикла событий. Чтобы это сделать, «под капотом» создается пара сокетов, которая используется, как своего рода сигнальный механизм. Когда работа выполнена пулом потоков, выполняется метод *_complete()* в классе. Этот метод формирует очередь из ожидающего коллбэка и результата перед записью первого байта в один из этих сокетов. Метод *fileno()* запрограммирован возвращать другой сокет. Когда этот байт записан, он даст циклу событий сигнал о том, что что-то произошло. Когда запускается метод *handle_receive()*, выполняются все функции обратного вызова для ранее отправленной работы. Если честно, от этого может закружиться голова.

Вот простой сервер, который демонстрирует использование пула потоков для выполнения длительных вычислений:

```
# Очень плохая реализация поиска чисел Фибоначчи
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

class UDPFibServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(128)
        n = int(msg)
        pool.run(fib, (n,), callback=lambda r: self.respond(r, addr))

    def respond(self, result, addr):
        self.sock.sendto(str(result).encode('ascii'), addr)

if __name__ == '__main__':
    pool = ThreadPoolHandler(16)
    handlers = [pool, UDPFibServer(('', 16000))]
    event_loop(handlers)
```

Чтобы опробовать сервер в работе, просто запустите его и поэкспериментируйте с другой программой на Python:

```
from socket import *
sock = socket(AF_INET, SOCK_DGRAM)
for x in range(40):
    sock.sendto(str(x).encode('ascii'), ('localhost', 16000))
    resp = sock.recvfrom(8192)
```

```
print(resp[0])
```

У вас должно получиться запустить эту программу многократно, из разных окон, и всё должно работать без остановки других программ, даже если всё будет работать медленнее и медленнее.

Должны ли вы использовать код из этого рецепта? Скорее всего, нет. Вместо этого вам стоит изучить более полный фреймворк, который решает ту же задачу. Однако если вы поймете базовые концепции, представленные здесь, вы поймете основные принципы, на базе которых работают такие фреймворки.

В качестве альтернативы программированию на базе коллбэков, управляемый событиями код часто использует корутины (сопрограммы). См. рецепт [12.12.](#), где приведён соответствующий пример.

11.13. Отсылка и получение больших массивов

Задача

Вы хотите отправлять и получать большие массивы смежных данных по сетевому соединению, делая настолько мало копий, насколько это возможно.

Решение

Следующие функции используют представления памяти (memoryviews) для отправки и получения больших массивов:

```
# zerocopy.py

def send_from(arr, dest):
    view = memoryview(arr).cast('B')
    while len(view):
        nsent = dest.send(view)
        view = view[nsent:]

def recv_into(arr, source):
    view = memoryview(arr).cast('B')
    while len(view):
        nrecv = source.recv_into(view)
        view = view[nrecv:]
```

Чтобы протестировать эту программу, сначала создайте сервер и клиентскую программу, соединенные через сокет. В сервере:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(('', 25000))
>>> s.listen(1)
>>> c,a = s.accept()
```

В клиенте (в отдельном интерпретаторе):

```
>>> from socket import *
>>> c = socket(AF_INET, SOCK_STREAM)
>>> c.connect(('localhost', 25000))
>>>
```

Вся идея этого рецепта в том, что вы можете пропихнуть огромный массив данных через соединение. В этом случае массивы могут быть созданы с помощью модуля *array* или *numpy*. Например:

```
# Сервер
>>> import numpy
>>> a = numpy.arange(0.0, 50000000.0)
>>> send_from(a, c)
>>>

# Клиент
>>> import numpy
>>> a = numpy.zeros(shape=50000000, dtype=float)
>>> a[0:10]
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> recv_into(a, c)
>>> a[0:10]
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>>
```

Обсуждение

Распределённые вычисления с большими объемами данных и параллельное программирование часто требуют получения/отправки больших кусков данных. Однако, чтобы это сделать, вам как-то нужно свернуть данные в сырье байты, которые можно использовать с низкоуровневыми сетевыми функциями. Вам также может потребоваться нарезать данные на куски

(чанки), поскольку большинство связанных с сетью функций не могут посыпать или посыпать огромные блоки данных целиком за раз.

Существует такой подход: как-то сериализовать данные — возможно, путём преобразования в байтовую строку. Однако это обычно кончается созданием копии данных. Даже если вы делаете это кусочек за кусочком, ваш код всё равно будет создавать кучу маленьких копий.

Этот рецепт обходит проблему путём хитрого фокуса с представлениями памяти (*memoryviews*). По сути, представление памяти — это наложение (*overlay*) существующего массива. Различные типы также могут быть «переколдованы» (приведены, *cast*) в представления памяти, чтобы позволить интерпретировать данные другим способом. В этом смысле следующего объявления:

```
view = memoryview(arr).cast('B')
```

Оно принимает массив *arr* и превращает в представление памяти (*memoryview*) беззнаковых байтов.

В этой форме представление (*view*) может быть передано связанным с сокетами функциям, таким как *sock.send()* или *send.recv_into()*. «Под капотом» эти методы могут работать напрямую с областью памяти. Например, *sock.send()* посылает данные напрямую из памяти, без копирования. *send.recv_into()* использует представление памяти (*memoryview*) как входной буфер для операции получения.

Оставшееся осложнение состоит в том, что функции сокетов могут работать только с разделёнными на части данными. В общем случае потребуется много различных вызовов *send()* и *recv_into()*, чтобы передать весь массив. Не волнуйтесь. После каждой операции представление (*view*) разрезается в соответствии с количеством отправленных или полученных байтов для создания нового представления. Новое представление также будет наложением на память (*memory overlay*). Так что никаких копий создаваться не будет.

Есть одна тонкость: получатель должен знать заранее, сколько данных будет отправлено, чтобы он мог либо заранее выделить массив, либо проверить, что он может принять данные в существующий массив. Если это важно в вашем случае, то посыпающий должен всегда сначала посыпать размер отправляемых данных, а уже потом — данные массива.

12. Конкурентное программирование

Python уже давно поддерживает различные подходы к конкурентному программированию, включая потоки, запуск подпроцессов, а также различные фокусы с использованием функций-генераторов. Рецепты этой главы будут относиться к различным аспектам конкурентного программирования, включая обычные приёмы использования потоков и подходы к параллельной обработке.

Опытным программистам известно, что конкурентное программирование склонно порождать проблемы. Поэтому главной темой этой главы будут рецепты, которые помогут создавать более надёжный и удобный для отладки код.

12.1. Запуск и остановка потоков

Задача

Вы хотите создавать и уничтожать потоки для конкурентного выполнения кода.

Решение

Библиотека *threading* может быть использована для выполнения любого вызываемого объекта Python в отдельном потоке. Чтобы сделать это, вы создаете экземпляр *Thread* и предоставляете ему вызываемый объект, который хотите выполнить. Вот простой пример:

```
# Код для выполнения в независимом потоке
import time

def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Создать и запустить поток
```

```
from threading import Thread
t = Thread(target=countdown, args=(10,))
t.start()
```

Когда вы создаёте экземпляр потока, он не начинает выполняться, пока вы не вызовете его метод `start()`, который вызывает целевую функцию с предоставленными вами аргументами.

Потоки выполняются в их собственных потоках системного уровня (т.е., потоке POSIX или потоке Windows), которые полностью управляются операционной системой. Будучи запущены, потоки выполняются независимо, пока целевая функция не вернёт результат. Вы можете опросить экземпляр потока, чтобы проверить, запущен ли он:

```
f t.is_alive():
    print('Still running')
else:
    print('Completed')
```

Вы также можете запросить объединение с потоком. Это означает, что поток, в котором выполнен этот вызов, будет ждать завершения потока, к которому применён метод:

```
t.join()
```

Интерпретатор остаётся запущенным до тех пор, пока все потоки не будут завершены. Для долго выполняющихся потоков или бэкграундных задач, которые запущены всё время, вам стоит попробовать создать поток «демоническим образом»:

```
t = Thread(target=countdown, args=(10,), daemon=True)
t.start()
```

Демонические потоки не могут быть объединены. Однако они автоматически уничтожаются, когда завершается главный поток.

Помимо двух показанных операций, есть не так уж много других вещей, которые вы можете сделать с потоками. Например, нет операций для завершения потока, отправки сигнала потоку, нельзя настроить расписание работы или выполнить любую другую высокоуровневую операцию. Если вам нужны эти возможности, вам придётся реализовать их самостоятельно.

Если вы хотите получить возможность завершать потоки, поток должен быть запрограммирован получать указания о выходе в определенных точках. Например вы можете поместить свой поток в такой класс:

```
class CountdownTask:
    def __init__(self):
        self._running = True

    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running and n > 0:
            print('T-minus', n)
            n -= 1
            time.sleep(5)

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
t.start()
...
c.terminate()      # Сигнал завершения
t.join()           # Ждать реального завершения (если необходимо)
```

Отправка запросов на завершение потоков может быть сложной в координации, если потоки выполняют блокирующие операции, такие как ввод-вывод. Например, если поток заблокирован на неопределенное время операцией ввода-вывода, он может никогда не вернуться, чтобы посмотреть, не убит ли он. Чтобы корректно разобраться с этим случаем, вам нужно аккуратно программировать потоки с использованием циклов с таймаутами. Например:

```
class IOTask:
    def terminate(self):
        self._running = False

    def run(self, sock):
        # sock is a socket
        sock.settimeout(5)          # Установить таймаут
        while self._running:
            # Выполнить блокирующую операцию ввода-вывода с таймаутом
            try:
                data = sock.recv(8192)
                break
            except socket.timeout:
                continue
        # Продолжение обработки
```

```
    ...
# Завершено
return
```

Обсуждение

Из-за глобальной блокировки интерпретатора (GIL) потоки Python ограничены моделью исполнения, которая разрешает только одному потоку выполняться в интерпретаторе в любой произвольно выбранный момент времени. По этой причине потоки Python не должны в общем случае использоваться для «тяжелых» вычислительных задач, где вам нужно добиться параллельного вычисления на нескольких процессорах. Они намного лучше подходят для обработки ввода-вывода или конкурентного выполнения в коде, производящего блокирующие операции (ожидание ввода-вывода, ожидание получения результатов из базы данных и т.п.)

Иногда вы можете встретить определение потоков через наследование от класса *Thread*. Например:

```
from threading import Thread

class CountdownThread(Thread):
    def __init__(self, n):
        super().__init__()
        self.n = 0

    def run(self):
        while self.n > 0:
            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

c = CountdownThread(5)
c.start()
```

Это работает, но вводит дополнительные зависимости между кодом и библиотекой *threading*. Поэтому вы можете использовать получившийся код только в контексте потоков, тогда как показанный ранее приём не имеет явной зависимости от *threading*. Путём освобождения вашего кода от таких зависимостей, он становится доступным для использования в других контекстах — с потоками или без. Например, вы сможете выполнить ваш код в отдельном процессе, используя модуль *multiprocessing*:

```
import multiprocessing
c = CountdownTask(5)
p = multiprocessing.Process(target=c.run)
p.start()
...
```

Ещё раз: это работает только в том случае, если класс *CountdownTask* был написан в нейтральной манере по отношению к средству обеспечения конкурентности (потоки, процессы и т.п.)

12.2. Как узнать, стартовал ли поток

Задача

Вы создали поток и хотите узнать, когда он стартует.

Решение

Ключевая возможность потоков — это то, что они выполняются независимо и недетерминистично. Это может вызвать сложности с синхронизацией, если другим потокам в программе перед выполнением следующих операций нужно знать, достиг ли некий поток некоторой точки в своём процессе выполнения. Чтобы решить эту проблему, используйте объект *Event* из библиотеки *threading*.

Экземпляры *Event* похожи на флаг “sticky”, который позволяет потокам ждать некого события. В начале событие установлено на 0. Если событие не установлено, и поток ждёт события, он будет заблокирован (т.е., уснёт) до тех пор, пока событие не будет установлено. Поток, который устанавливает событие, разбудит все потоки, которые находятся в состоянии ожидания (если они есть). Если поток настроен ждать события, которое уже установлено, он просто продолжит выполнение.

Вот пример кода, использующего *Event* для координации запуска потоков:

```
from threading import Thread, Event
import time

# Код для выполнения в независимом потоке
def countdown(n, started_evt):
    print('countdown starting')
    time.sleep(n)
    started_evt.set()
```

```

started_evt.set()
while n > 0:
    print('T-minus', n)
    n -= 1
    time.sleep(5)

# Создать объект события, который будет использован для сигнала о запуске
started_evt = Event()

# Запустить поток и передать событие запуска
print('Launching countdown')
t = Thread(target=countdown, args=(10,started_evt))
t.start()

# Ждать запуска потока
started_evt.wait()
print('countdown is running')

```

Когда вы запустите этот код, сообщение “countdown is running” появится после сообщения “countdown starting”. Это координируется путём события, которое заставляет главный поток ждать, пока функция `countdown()` не напечатает сообщение о запуске.

Обсуждение

Объекты `Event` наилучшим образом подходят для одноразовых событий. Вы создаёте событие, потоки ждут установки события, и когда оно установлено, `Event` отбрасывается. Хотя можно очистить событие, используя его метод `clear()`, безопасная очистка события и ожидание его повторной установки трудно координируется, а также может привести к пропущенным событиям, дедлокам или другим проблемам (в частности, вы не можете гарантировать, что запрос на очистку события после его установки будет выполнен до высвобождения циклов потоков для ожидания события).

Если поток собирается снова и снова сигнализировать событием, вам лучше подойдёт объект `Condition`. Например, этот код реализует периодический таймер, который могут отслеживать другие потоки — и видеть, что время истекло:

```

import threading
import time

class PeriodicTimer:
    def __init__(self, interval):
        self.interval = interval

```

```

    self._flag = 0
    self._cv = threading.Condition()

def start(self):
    t = threading.Thread(target=self.run)
    t.daemon = True
    t.start()

def run(self):
    """
    Запустить таймер и уведомлять ждущие потоки
    после каждого интервала
    """

    while True:
        time.sleep(self._interval)
        with self._cv:
            self._flag ^= 1
            self._cv.notify_all()

def wait_for_tick(self):
    """
    Ждать следующего срабатывания таймера
    """

    with self._cv:
        last_flag = self._flag
        while last_flag == self._flag:
            self._cv.wait()

# Пример использования таймера
ptimer = PeriodicTimer(5)
ptimer.start()

# Два потока, синхронизирующихся по таймеру
def countdown(nticks):
    while nticks > 0:
        ptimer.wait_for_tick()
        print('T-minus', nticks)
        nticks -= 1

def countup(last):
    n = 0
    while n < last:
        ptimer.wait_for_tick()
        print('Counting', n)
        n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()

```

Важнейшая возможность объектов *Event* в том, что они пробуждают все

ждущие объекты. Если вы пишете программу, в которой вам просто нужна возможность пробуждать единственный ждущий поток, то, вероятно, лучше будет использовать объект *Semaphore* или *Condition*.

Например, рассмотрим такой код, использующий семафоры:

```
# Поток-воркер
def worker(n, sema):
    # Ждёт сигнала
    sema.acquire()
    # Выполняет работу
    print('Working', n)

# Создаем несколько потоков
sema = threading.Semaphore(0)
nworkers = 10
for n in range(nworkers):
    t = threading.Thread(target=worker, args=(n, sema,))
    t.start()
```

Если вы запустите эту программу, стартует пул потоков, но ничего не произойдёт, поскольку все они заблокированы, ожидая получения семафора. Каждый раз, когда высвобождается семафор, только один воркер проснется и запустится. Например:

```
>>> sema.release()
Working 0
>>> sema.release()
Working 1
>>>
```

Написание кода, который использует хитрую синхронизацию между потоками, может взорвать вашу голову. Более разумный подход к использованию потоков — работать с ними как с коммуницирующими задачами, используя очереди (или же как с акторами). Очереди описаны в следующем рецепте. Акторы описаны в [рецепте 12.10](#).

12.3. Коммуникация между потоками

Задача

В вашей программе есть несколько потоков, и вы хотите безопасно

коммуницировать или обмениваться данными между ними.

Решение

Вероятно, самый безопасный путь переслать данные из одного потока в другой — это использовать *Queue* из библиотеки *queue*. Чтобы сделать это, создайте экземпляр *Queue*, который будет общим для всех потоков. Затем потоки должны использовать операции *put()* или *get()*, чтобы добавлять или убирать элементы из очереди. Например:

```
from queue import Queue
from threading import Thread

# Поток, который производит данные
def producer(out_q):
    while True:
        # Производим данные
        ...
        out_q.put(data)

# Поток, который потребляет данные
def consumer(in_q):
    while True:
        # Получаем данные
        data = in_q.get()
        # Обрабатываем данные
        ...

# Создаем разделяемую (shared) очередь и запускаем оба потока
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Экземпляры *Queue* уже имеют все нужные блокировки, поэтому они могут безопасно стать общими для любого количества потоков.

При использовании очередей иногда довольно сложно скоординировать отключение продюсера (производителя) и консьюмера (потребителя). Общепринятое решение этой проблемы опирается на специальное «сторожевое» значение, которое при помещении в очередь заставляет консьюмеров завершаться. Например:

```
from queue import Queue
```

```

from threading import Thread

# Объект, который сигнализирует об отключении
_sentinel = object()

# Поток, который производит данные
def producer(out_q):
    while running:
        # Производим данные
        ...
        out_q.put(data)

    # Поместить стража в очередь, чтобы сигнализировать о завершении
    out_q.put(_sentinel)

# Поток, который потребляет данные
def consumer(in_q):
    while True:
        # Получаем данные
        data = in_q.get()

        # Проверяем на предмет сигнала о завершении
        if data is _sentinel:
            in_q.put(_sentinel)
            break

        # Обрабатываем данные
        ...

```

В этом примере есть тонкий момент — при получении специального «сторожевого» значения консьюмер немедленно помещает его обратно в очередь. Это позволяет «сторожевому» значению распространиться и попасть к другим консьюмерам, подключенными к той же очереди — и это отключит их, один за одним.

Хотя очереди — это наиболее распространённый механизм коммуникации, вы можете построить собственные структуры данных, которые просто должны иметь требуемую блокировку и синхронизацию. Самый обычный способ это сделать — обернуть ваши структуры данных в условную переменную. Например, вот так вы можете построить потокобезопасную очередь с приоритетом, обсуждавшуюся в [рецепте 1.5.](#):

```

import heapq
import threading

class PriorityQueue:
    def __init__(self):

```

```

    self._queue = []
    self._count = 0
    self._cv = threading.Condition()

    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._queue, (-priority, self._count, item))
            self._count += 1
            self._cv.notify()

    def get(self):
        with self._cv:
            while len(self._queue) == 0:
                self._cv.wait()
            return heapq.heappop(self._queue)[-1]

```

Коммуникация потоков через очередь — односторонний и недетерминистический процесс. В общем случае нет возможности узнать, когда получающий поток в действительности получил и обработал сообщение. Однако объекты *Queue* предоставляют некоторые базовые возможности завершения, как показано в методах *task_done()* и *join()* в этом примере:

```

from queue import Queue
from threading import Thread

# Поток, который производит данные
def producer(out_q):
    while running:
        # Произвести данные
        ...
        out_q.put(data)

# Поток, который потребляет данные
def consumer(in_q):
    while True:
        # Получаем данные
        data = in_q.get()
        # Обрабатываем данные
        ...
        # Сигнализируем о завершении
        in_q.task_done()

# Создаём разделяемую (shared) очередь и запускаем оба потока
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()

```

```
t2.start()
# Ждём, пока все произведённые элементы будут потреблены
q.join()
```

Если потоку нужно немедленно узнавать о том, что поток-консьюмер обработал определённый элемент данных, вы должны «спарить» отправленные данные с объектом *Event*, что позволит продюсеру отслеживать прогресс. Например:

```
from queue import Queue
from threading import Thread, Event

# Поток, который производит данные
def producer(out_q):
    while running:
        # Производим данные
        ...
        # Создаём пару (data, event) и передаём её потребителю
        evt = Event()
        out_q.put((data, evt))
        ...
        # Ждём, пока потребитель не обработает данные
        evt.wait()

# Поток, который потребляет данные
def consumer(in_q):
    while True:
        # Получаем данные
        data, evt = in_q.get()
        # Обрабатываем данные
        ...
        # Сигнализируем о завершении
        evt.set()
```

Обсуждение

Написание потоковых программ, основанных на простых очередях, часто будет хорошим способом сохранить рассудок. Если можете разбить всё на простые потокобезопасные очереди, то вы обнаружите, что вам не нужно замусоривать вашу программу блокировками и прочей низкоуровневой синхронизацией. Также коммуникация с помощью очередей часто помогает создать масштабируемые проекты, которые можно потом перевести на другие шаблоны коммуникации на базе сообщений. Например, вы могли бы без переделки внутренней архитектуры очередей разделить вашу программу на множество процессов или даже превратить в распределённую систему.

Предостережение тем, кто будет использовать потоковые очереди: помещение элемента в очередь не означает, что элемент будет скопирован. Поэтому коммуникация в действительности означает передачу ссылки на объект между потоками. Если вы беспокоитесь о появлении общего состояния (shared state), имеет смысл передавать только неизменяемые структуры данных (целые числа, строки или кортежи), или же делать «глубокие копии» помещаемых в очередь элементов. Например:

```
from queue import Queue
from threading import Thread
import copy

# Поток, который производит данные
def producer(out_q):
    while True:
        # Производим данные
        ...
        out_q.put(copy.deepcopy(data))

# Поток, который потребляет данные
def consumer(in_q):
    while True:
        # Получаем данные
        data = in_q.get()
        # Обрабатываем данные
        ...

```

Объекты `Queue` предоставляют несколько дополнительных возможностей, которые могут быть полезны в некоторых контекстах. Если вы создаёте `Queue`, задавая размер — `Queue(N)` (что необязательно), — то в очередь можно будет поместить ограниченное количество элементов, прежде чем `put()` заблокирует продюсера. Определение верхней границы количества элементов в очереди может иметь смысл, если есть разница в скорости работы продюсера и консьюмера. Например, продюсер может генерировать элементы гораздо быстрее, чем консьюмер может их обработать. С другой стороны, если очередь блокируется при заполнении, то это может вызвать непредумышленный каскадный эффект, который распространится на всю программу и вызовет дедлок или сбои в работе. В общем, проблема передачи потока управления между общающимися потоками намного сложнее, чем кажется. Если вы внезапно обнаружите, что пытаетесь решить проблему за счёт манипуляций с размерами очередей, то это признак хрупкого дизайна или каких-то других врождённых проблем с масштабированием.

И метод `get()`, и метод `put()` поддерживают неблокируемость и таймауты.

Например:

```
import queue

q = queue.Queue()

try:
    data = q.get(block=False)
except queue.Empty:
    ...

try:
    q.put(item, block=False)
except queue.Full:
    ...

try:
    data = q.get(timeout=5.0)
except queue.Empty:
    ...
```

Обе эти опции могут быть использованы для избежания проблемы неопределённого блокирования при какой-либо операции с очередью.

Например, неблокирующий `put()` может быть использован с очередью фиксированного размера, чтобы реализовать различные типы обрабатывающего кода, который будет срабатывать при заполнении очереди.

Например, можно делать запись в лог и отбрасывать элемент:

```
def producer(q):
    ...
    try:
        q.put(item, block=False)
    except queue.Full:
        log.warning('queued item %r discarded!', item)
```

Таймауты полезны, если вы пытаетесь заставить потоки-консьюмеры периодически осуществлять операции типа `q.get()`, чтобы они могли проверить такие вещи, как флаг завершения (описано в **рецепте 12.1.**):

```
_running = True
def consumer(q):
    while _running:
        try:
            item = q.get(timeout=5.0)
            # Обрабатываем элемент
            ...
        except queue.Empty:
```

```
pass
```

И последнее: есть методы `q.qsize()`, `q.full()` и `q.empty()`, которые позволяют узнать текущий размер и статус очереди. Однако стоит знать, что все они ненадёжны в многопоточном окружении. Например, вызов `q.empty()` может сообщить вам, что очередь пуста, но за то время, которое пройдёт от момента вызова, другой поток может поместить элементы в очередь. Лучше писать код, который не полагается на такие функции.

12.4. Блокировка критически важных участков

Задача

Ваша программа использует потоки, и вы хотите заблокировать критически важные участки кода, чтобы избежать состояния гонки (*race condition*).

Решение

Чтобы сделать изменяемые объекты безопасными для использования в многопоточной программе, используйте объекты `Lock` из библиотеки `threading`, как показано ниже:

```
import threading

class SharedCounter:
    ...

    Объект счётчика, который может быть общим (shared) для нескольких потоков.
    ...

    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        ...
        Инкрементирует счётчик с блокировкой.
        ...
        with self._value_lock:
            self._value += delta

    def decr(self,delta=1):
        ...
        Декрементирует счётчик с блокировкой.
```

```
    ...
    with self._value_lock:
        self._value -= delta
```

Lock гарантирует взаимное исключение при использовании с инструкцией *with* — это значит, что только одному потоку за раз разрешено исполнять инструкции в теле блока *with*. Инструкция *with* получает блокировку на всё время выполнения находящихся под ней инструкций, и освобождает блокировку, когда поток управления покидает выделенный отступом блок инструкций.

Обсуждение

Расписание потоков является врождённо недетерминистичным. По этой причине неудача с использованием блокировок в основанных на потоках программах может привести к порче данных и странному поведению, известному как «состояние гонки» (race condition). Чтобы избежать этого, блокировки должны быть применены во всех случаях, когда к изменяемому состоянию (mutable state) осуществляется доступ со стороны многих потоков.

В старом коде Python часто можно увидеть явное получение и освобождение блокировок. Например, вот вариант последнего примера:

```
import threading

class SharedCounter:
    ...

    Объект счётчика, который может быть общим (shared) для нескольких потоков.

    ...

    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        ...

        Инкрементирует счётчик с блокировкой.

        ...

        self._value_lock.acquire()
        self._value += delta
        self._value_lock.release()

    def decr(self,delta=1):
        ...

        Декрементирует счётчик с блокировкой.

        ...
```

```
    self._value_lock.acquire()
    self._value -= delta
    self._value_lock.release()
```

Инструкция *with* более элегантна и менее подвержена ошибкам — особенно в ситуациях, когда программист может забыть вызвать метод *release()*, или если в программе возбуждается исключение во время удерживания блокировки (а инструкция *with* гарантирует, что блокировки будут освобождены в обоих этих случаях).

Чтобы избежать потенциального дедлока, программы, использующие блокировки, должны быть написаны таким образом, чтобы каждому потоку разрешалось получать только одну блокировку за раз. Если это невозможно, вам может потребоваться добавить более продвинутую схему избежания дедлоков, как описано в [рецепте 12.5](#).

В библиотеке *threading* вы найдёте другие примитивы синхронизации, такие как *RLock* и объекты *Semaphore*. Опыт показывает, что они имеют более специальное назначение и не должны быть использованы для простых блокировок изменяемого состояния. *RLock* (объект многократной блокировки) — это блокировка, которая может приобретаться несколько раз одним и тем же потоком. В основном она используется для реализации, основанной на коде блокировки (code based locking) или синхронизации, основанной на конструкции, известной как «монитор». С блокировкой такого типа только одному потоку разрешается использовать функцию или методы класса, пока блокировка удерживается. Например, вы можете реализовать класс *SharedCounter* таким образом:

```
import threading

class SharedCounter:
    """
    Объект счётчика, который может быть общим (shared) для нескольких потоков.
    """

    _lock = threading.RLock()

    def __init__(self, initial_value = 0):
        self._value = initial_value

    def incr(self, delta=1):
        """
        Инкрементирует счётчик с блокировкой.
        """

        with SharedCounter._lock:
            self._value += delta
```

```
def decr(self, delta=1):
    """
    Декрементирует счётчик с блокировкой.
    """

    with SharedCounter._lock:
        self.incr(-delta)
```

В этом варианте кода имеется единственная блокировка уровня класса, разделяемая всеми экземплярами класса. Вместо привязки блокировки к изменяемому состоянию каждого экземпляра, эта блокировка предназначена для синхронизации методов класса. В частности, эта блокировка позволяет убедиться, что только одному потоку в конкретный момент времени разрешено использовать методы класса. Однако, в отличие от стандартной блокировки, методы, которые получили блокировку, могут вызывать другие методы, которые также используют блокировку (например, см. метод `decr()`).

В этой реализации есть такая возможность: создаётся только одна блокировка, независимо от количества созданных экземпляров счётчика. Поэтому она намного более эффективно использует память в ситуациях, когда создается большое количество счётчиков. Однако в этом есть и потенциальный недостаток: это может вызвать серьёзную борьбу за блокировку в программах, использующих большое количество потоков и часто совершающих обновления счётчиков.

Объект `Semaphore` — это примитив синхронизации, основанный на разделяемом (shared) счётчике. Если счётчик ненулевой, инструкция `with` уменьшает его значение, и потоку разрешается работать. Счётчик инкрементируется после завершения выполнения блока `with`. Если счётчик равен нулю, работа блокируется, пока счётчик не увеличится другим потоком. Хотя семафор может быть использован так же, как и стандартный `Lock`, добавленная сложность в реализации отрицательно влияет на производительность. Объекты `Semaphore` более полезны в качестве замены простой блокировки в приложениях, использующих обмен сигналами между потоками или троттлинг (throttling, дросселирование). Например, если вы хотите ограничить конкурентность в каком-то конкретном участке кода, то вы можете использовать семафор:

```
from threading import Semaphore
import urllib.request

# Самое большее 5 потоков могут выполняться одновременно
```

```
_fetch_url_sema = Semaphore(5)
def fetch_url(url):
    with _fetch_url_sema:
        return urllib.request.urlopen(url)
```

Если вы заинтересованы в теории и реализации потоковых примитивов синхронизации, обратитесь к любому учебнику по операционным системам.

12.5. Блокировка с избежанием дедлока

Задача

Вы пишете многопоточную программу, где потоки должны получать более чем одну блокировку за раз, избегая дедлока.

Решение

В многопоточных программах обычной причиной возникновения дедлока являются потоки, которые пытаются получить несколько блокировок за раз. Например, если поток получает первую блокировку, но затем блокируется, пытаясь получить вторую блокировку, этот поток может заблокировать выполнение других потоков и заставить программу приостановиться.

Одно из решений для избежания дедлока — присвоить каждой блокировке в программе уникальный номер и заставить соблюдать выполнение правила приобретения по порядку: несколько блокировок могут быть получены только по возрастанию номеров. Это очень легко реализовать с помощью менеджера контекста:

```
import threading
from contextlib import contextmanager

# Локальное для потока состояние для хранения информации
# об уже полученных блокировках
_local = threading.local()

@contextmanager
def acquire(*locks):
    # Сортирует блокировки по идентификатору объекта
    locks = sorted(locks, key=lambda x: id(x))

    # Убеждается, что порядок блокировки ранее
    # приобретённых блокировок не нарушен
```

```

acquired = getattr(_local, 'acquired', [])
if acquired and max(id(lock) for lock in acquired) >= id(locks[0]):
    raise RuntimeError('Lock Order Violation')

# Получает все блокировки
acquired.extend(locks)
_local.acquired = acquired

try:
    for lock in locks:
        lock.acquire()
        yield
finally:
    # Освобождает блокировки в порядке, обратном их получению
    for lock in reversed(locks):
        lock.release()
    del acquired[-len(locks):]

```

Чтобы использовать этот менеджер контекста, вы просто выделяете объекты блокировок обычным образом, но используете функцию `acquire()`, когда хотите работать с одной или более блокировками. Например:

```

import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock, y_lock):
            print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock, x_lock):
            print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()

```

Если вы запустите эту программу, то обнаружите, что она отлично работает без остановки и дедлока — даже если приобретение блокировок определено в каждой функции в разном порядке.

Основной секрет этого рецепта — в первой инструкции, которая сортирует блокировки в соответствии с идентификатором объекта. Путём сортировки блокировок мы добиваемся того, что они всегда приобретаются в строгом порядке, независимо от того, как пользователь может передать их в `acquire()`.

Решение использует локальное для каждого потока хранилище, чтобы решить тонкую проблему с определением потенциального дедлока при нескольких вложенных операциях `acquire()`. Предположим, например, что вы пишете такой код:

```
import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock):
            with acquire(y_lock):
                print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock):
            with acquire(x_lock):
                print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()
```

Если вы запустите эту версию программы, один из потоков упадёт с исключением:

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/local/lib/python3.3/threading.py", line 639, in _bootstrap_
    self.run()
  File "/usr/local/lib/python3.3/threading.py", line 596, in run
    self._target(*self._args, **self._kwargs)
  File "deadlock.py", line 49, in thread_1
    with acquire(y_lock):
  File "/usr/local/lib/python3.3/contextlib.py", line 48, in __enter__
    return next(self.gen)
```

```
File "deadlock.py", line 15, in acquire
    raise RuntimeError("Lock Order Violation")
RuntimeError: Lock Order Violation
>>>
```

Это падение вызвано тем, что каждый поток запоминает блокировку, которую он уже приобрёл. Функция `acquire()` проверяет список ранее приобретённых блокировок и принудительно внедряет ограничение упорядочивания: приобретённые ранее блокировки должны иметь ID объекта, который меньше, чем у новых приобретаемых блокировок.

Обсуждение

Проблема дедлока в программах с использованием потоков широко известна (это общая тема для всех учебников по операционным системам).

Эмпирическое правило: пока вы можете удостовериться, что потоки могут удерживать только одну блокировку в данный момент времени, дедлока в вашей программе не будет. Однако если приобретаются несколько блокировок, то всё меняется.

Поиск дедлока и восстановление после него — крайне хитрая проблема с небольшим количеством элегантных решений. Например, обычная схема поиска дедлока и восстановления работоспособности подразумевает использование таймера типа «сторожевой пёс». Пока потоки выполняются, они периодически сбрасывают таймер, и пока это работает гладко, всё хорошо. Однако если возникает дедлок, «сторожевой пёс» достигнет порогового значения. В этой точке программа выполняет восстановление, убивая себя и перезапускаясь.

Еще одна стратегия избежания дедлока — это когда операции блокировки выполняются таким образом, что просто не позволяют программе войти в состояние дедлока. Может быть математически доказано, что решение, при котором блокировки приобретаются в строгом порядке увеличения номеров ID объектов, позволяет избежать дедлока. Доказательство мы оставляем читателю (суть в том, что получение блокировок строго в возрастающем порядке позволяет избежать циклических зависимостей блокировок, а это является необходимым условием возникновения дедлока).

В качестве последнего примера приведём классическую задачу о дедлокае, которая называется «проблема обедающих философов». В этой задаче пять

философов сидят вокруг стола, на котором стоят пять мисок риса и лежат пять палочек для еды. Каждый философ представляет независимый поток, а каждая палочка — блокировку. В задаче философы либо размышляют, либо едят. Но чтобы поесть риса, философу нужны две палочки. К сожалению, если все философы протянут руку и возьмут палочку слева от себя, все они останутся с одной палочкой и умрут от голода. Ужасная картина.

Используя приведённое решение, мы можем написать простую реализацию проблемы пяти философов, избавленную от дедлока:

```
import threading

# Поток-философ
def philosopher(left, right):
    while True:
        with acquire(left,right):
            print(threading.currentThread(), 'eating')

# Палочки (представлены блокировками)
NSTICKS = 5
chopsticks = [threading.Lock() for n in range(NSTICKS)]

# Создаём всех философов
for n in range(NSTICKS):
    t = threading.Thread(target=philosopher,
                         args=(chopsticks[n],
                                chopsticks[(n+1) % NSTICKS]))
    t.start()
```

И последнее: стоит отметить, что для избежания дедлока все блокирующие операции должны выполняться с использованием нашей функции `acquire()`. Если какой-то фрагмент кода решит приобрести блокировку напрямую, алгоритм избежания дедлока перестанет работать.

12.6. Хранение «потокоспецифичного» состояния

Задача

Вам нужно хранить состояние, которое специфично для выполняющегося в данный момент потока и невидимо для других потоков.

Решение

Иногда в многопоточных программах вам нужно хранить данные, которые специфичны для текущего выполняющегося потока. Чтобы сделать это, создайте локальное для потока хранилище с помощью `threading.local()`.

Атрибуты, которые сохраняются в этот объект и читаются из него, доступны только выполняющемуся в данный момент потоку, но не другим.

Интересный практический пример использования локального для потока хранилища — менеджер контекста класса `LazyConnection`, который мы определяли в [рецепте 8.3](#). Вот его слегка модифицированная версия, которая безопасно работает с многими потоками:

```
from socket import socket, AF_INET, SOCK_STREAM
import threading

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.local = threading.local()

    def __enter__(self):
        if hasattr(self.local, 'sock'):
            raise RuntimeError('Already connected')
        self.local.sock = socket(self.family, self.type)
        self.local.sock.connect(self.address)
        return self.local.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.local.sock.close()
        del self.local.sock
```

Здесь внимательно понаблюдайте, как используется атрибут `self.local`. Он инициализируется как экземпляр `threading.local()`. Другие методы затем манипулируют сокетом, который хранится как `self.local.sock`. Этого достаточно, чтобы сделать возможным использование экземпляра `LazyConnection` в нескольких потоках. Например:

```
from functools import partial

def test(conn):
    with conn as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
```

```
s.send(b'Host: www.python.org\r\n')
s.send(b'\r\n')
resp = b''.join(iter(partial(s.recv, 8192), b''))

print('Got {} bytes'.format(len(resp)))

if __name__ == '__main__':
    conn = LazyConnection(('www.python.org', 80))

    t1 = threading.Thread(target=test, args=(conn,))
    t2 = threading.Thread(target=test, args=(conn,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

Причина, по которой это работает, в том, что каждый поток на самом деле создает собственное выделенное соединение через сокет (хранящееся как *self.local.sock*). Так что когда разные потоки производят операции с сокетом, они не мешают друг другу, поскольку выполняются над разными сокетами.

Обсуждение

Создание и манипулирование специфичным для каждого потока состоянием — это не та проблема, которая возникает часто. Однако когда она всё же возникает, это обычно ситуация, где используемый различными потоками объект должен манипулировать неким выделенным системным ресурсом, таким как сокет или файл. Вы не можете просто иметь один объект сокета, разделяемый между всеми, потому что возникнет хаос, когда несколько потоков будут одновременно читать и писать в этот сокет. Локальное для каждого потока хранилище исправляет это путём того, что делает эти ресурсы видимыми только тому потоку, где они используются.

В этом рецепте использование *threading.local()* заставляет класс *LazyConnection* поддерживать одно соединение на поток, что противоположно одному соединению на весь процесс. Это тонкое, но интересное отличие.

«Под капотом» экземпляр *threading.local()* поддерживает отдельный экземпляр словаря на каждый поток. Все обычные операции над экземплярами, такие как получение, присваивание, удаление значения просто манипулируют словарём конкретного потока. То, что каждый поток использует отдельный словарь, обеспечивает изоляцию данных.

12.7. Создание пула потоков

Задача

Вы хотите создать пул потоков-воркеров для обслуживания клиентов или выполнения других типов работы.

Решение

В библиотеке `concurrent.futures` есть класс `ThreadPoolExecutor`, который можно использовать для этой цели. Вот пример простого TCP-сервера, использующего пул потоков для обслуживания клиентов:

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_client(sock, client_addr):
    ...
    Обрабатывает клиентское соединение
    ...
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr):
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

Если вы хотите вручную создать собственный пул потоков, обычно достаточно просто это сделать с помощью `Queue`. Вот немного отличающаяся «ручная» реализация того же кода:

```

from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_client(q):
    """
    Обрабатывает клиентское соединение
    """

    sock, client_addr = q.get()
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr, nworkers):
    # Запускаем клиентских воркеров
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Запускаем сервер
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server('', 15000), 128)

```

Преимущество *ThreadPoolExecutor* перед ручной реализацией в том, что он облегчает отправляющему (сабмиттеру) задачу приёмки результатов из вызванной функции. Например, вы можете написать такой код:

```

from concurrent.futures import ThreadPoolExecutor
import urllib.request

def fetch_url(url):
    u = urllib.request.urlopen(url)
    data = u.read()
    return data

pool = ThreadPoolExecutor(10)

```

```
# Отправить работу в пул
a = pool.submit(fetch_url, 'http://www.python.org')
b = pool.submit(fetch_url, 'http://www.pypy.org')

# Получить результаты
x = a.result()
y = b.result()
```

Объекты результатов в примере обрабатывают всё блокирование и выполняют координацию, необходимую для получения данных обратно из потока-воркера. Говоря конкретно, операция `a.result()` блокируется до тех пор, пока соответствующая функция не выполнится пулом и не вернёт значение.

Обсуждение

В общем случае вам стоит избегать написания программ, которые позволяют количеству потоков расти неограниченно. Например, взгляните на такой сервер:

```
from threading import Thread
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(sock, client_addr):
    ...

    Обрабатывает клиентское соединение
    ...

    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr, nworkers):
    # Запускаем сервер
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        t = Thread(target=echo_client, args=(client_sock, client_addr))
        t.daemon = True
        t.start()
```

```
echo_server(' ', 15000)
```

Хотя это работает, но не запрещает некому асинхронному хипстеру атаковать ваш сервер, заставив его создавать так много потоков, что заставит программу потребить все ресурсы и упасть. Путём использования предварительно инициализированного пула потоков вы аккуратно создаете верхнюю границу объема поддерживаемой конкурентности.

Вы можете быть обеспокоены эффектом, который произведёт создание большого количества потоков. Однако современные системы легко создают пулы из нескольких тысяч потоков. Более того, тысяча просто ждущих работы потоков не оказывает серьёзного (а то и никакого) воздействия на производительность другого кода (спящий поток ничего не делает). Конечно, если все эти потоки проснутся одновременно и начнут требовать CPU, это будет совсем другая история — особенно в свете глобальной блокировки интерпретатора (GIL). Общее правило: используйте пулы потоков для связанной со вводом-выводом обработки.

Возможно, что при создании больших пулов потоков у вас возникнет опасение по поводу потребления памяти. Например, если вы создаёте 2000 потоков на OS X, система покажет, что процесс Python использует 9 гигабайт виртуальной памяти. Но не дайте этим цифрам себя запутать. При создании потока операционная система резервирует область виртуальной памяти, чтобы держать в ней стек выполнения потока (часто до 8 мегабайт). Однако только небольшой фрагмент этой памяти на самом деле отображен на реальную память. Если вы взглянете на вопрос повнимательнее, вы обнаружите, что интерпретатор Python использует намного меньше реальной памяти (для 2000 потоков используется только 70 мегабайт, но не 9 гигабайт). Если объем занятой виртуальной памяти важен, вы можете снизить его, используя функцию *threading.stack_size()*. Например:

```
import threading  
threading.stack_size(65536)
```

Если вы добавите этот вызов и повторите эксперимент с созданием 2000 потоков, то процесс Python съест всего 210 мегабайт виртуальной памяти, хотя объем реальной памяти останется практически таким же. Обратите внимание, что размер стека потока должен быть не меньше 32768 байтов, и обычно это значение является результатом умножения целого числа на размер системной страницы памяти (4096, 8192 и т.п.)

12.8. Простое параллельное программирование

Задача

У вас есть программа, которая выполняет много работы, нагружающей CPU, и вы хотите заставить её работать быстрее путём использования нескольких CPU.

Решение

Библиотека `concurrent.futures` предоставляет класс `ProcessPoolExecutor`, который может быть использован для выполнения тяжелых вычислительных задач в отдельно запущенных экземплярах интерпретатора Python.

Для использования этого класса вам сначала нужно иметь, что вычислять. Давайте проиллюстрируем это с помощью простого практического примера.

Предположим, что у вас есть каталог со сжатыми gzip логами веб-сервера Apache:

```
logs/
  20120701.log.gz
  20120702.log.gz
  20120703.log.gz
  20120704.log.gz
  20120705.log.gz
  20120706.log.gz
  ...
  ...
```

Предположим также, что каждый файл содержит такие строки:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 3
...
...
```

Вот простой скрипт, который берёт эти данные и находит все хосты, которые обращались к файлу `robots.txt`:

```

# findrobots.py

import gzip
import io
import glob

def find_robots(filename):
    """
    Находит в одном лог-файле всех хостов, который обращались к robots.txt
    """

    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f, encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    """
    Находит всех хостов во всех файлах
    """

    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    for robots in map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)

```

Приведённая выше программа написана в популярном стиле map-reduce. Функция *find_robots()* применяется к коллекции имён файлов, и результаты объединяются в один результат (множество *all_robots* в функции *find_all_robots()*).

Теперь предположим, что вам нужно изменить эту программу таким образом, чтобы использовать несколько CPU. Оказывается, это достаточно просто — достаточно заменить операцию *map()* на похожую операцию из библиотеки *concurrent.futures*, выполняемую пулом процессов. Вот слегка модифицированная версия:

```

# findrobots.py

import gzip

```

```

import io
import glob
from concurrent import futures

def find_robots(filename):
    """
    Находит в одном лог-файле всех хостов, который обращались к robots.txt
    """

    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f,encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    """
    Находит всех хостов во всех файлах
    """

    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    with futures.ProcessPoolExecutor() as pool:
        for robots in pool.map(find_robots, files):
            all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)

```

Теперь скрипт производит тот же самый результат, но работает в 3,5 раза быстрее на четырёхядерном компьютере. Реальная производительность будет варьироваться в зависимости от количества CPU, доступных на вашем компьютере.

Обсуждение

Обычно *ProcessPoolExecutor* используют так:

```

from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor() as pool:
    ...
    do work in parallel using pool
    ...

```

«Под капотом» *ProcessPoolExecutor* создаёт N независимо работающих интерпретаторов Python, где N — это количество доступных обнаруженных в системе CPU. Вы можете изменить количество создаваемых процессов, предоставив необязательный аргумент (*ProcessPoolExecutor(N)*). Пул работает до тех пор, пока не будет выполнена последняя инструкция в блоке *with*, после чего пул процессов завершается. Однако программа будет ждать, пока вся отправленная работа не будет сделана.

Работа, которая отправляется в пул, должна быть определена в форме функции. Есть два метода отправки. Если вы пытаетесь распараллелить генератор списка или операцию *map()*, используйте *pool.map()*:

```
# Функция, которая делает много работы
def work(x):
    ...
    return result

# Непараллельный код
results = map(work, data)

# Параллельная реализация
with ProcessPoolExecutor() as pool:
    results = pool.map(work, data)
```

Или же вы можете вручную отправлять (сабмитить) единичные задачи, используя метод *pool.submit()*:

```
# Какая-то функция
def work(x):
    ...
    return result

with ProcessPoolExecutor() as pool:
    ...
    # Пример отправки работы в пул
    future_result = pool.submit(work, arg)

    # Получение результата (блокирует до завершения)
    r = future_result.result()
    ...
```

Если вы отправляете работу вручную, результатом будет экземпляр *Future*. Чтобы получить реальный результат, вы вызываете его метод *result()*. Это вызывает блокировку на время, пока результат не посчитается и не будет возвращён пулом.

Вместо блокировки вы можете сделать так, чтобы по завершению работы выполнялся коллбэк. Например:

```
def when_done(r):
    print('Got:', r.result())

with ProcessPoolExecutor() as pool:
    future_result = pool.submit(work, arg)
    future_result.add_done_callback(when_done)
```

Предоставленная пользователем функция обратного вызова (коллбэк) получает экземпляр *Future*, который должен быть использован для получения реального результата (путём вызова его метода *result()*).

Хотя пулы процессов могут быть простыми в использовании, есть несколько важных вопросов, которые нужно держать в голове при написании крупных программ. Приведем их без какого-либо упорядочивания:

- Этот приём распараллеливания хорошо работает только для задач, которые легко раскладываются на независимые части.
- Работа должна отправляться (сабмититься) в форме простых функций. Параллельное выполнение методов экземпляров, замыканий или других конструкций не поддерживается.
- Аргументы функций и возвращаемые значения должны быть совместимы с *pickle*. Работа выполняется в отдельном интерпретаторе при использовании межпроцессной коммуникации. Так что данные, которыми обмениваются интерпретаторы, должны сериализоваться.
- Функции, отправляемые на выполнение, не должны поддерживать постоянное состояние или иметь побочные эффекты. За исключением простых вещей типа логирования, у вас нет никакого контроля над поведением процессов-потомков после их запуска. Так что для сохранения рассудка лучше поддерживать простоту программы и выполнять работу в чистых функциях, которые не изменяют своё окружение.
- Пулы процессов в Unix создаются с помощью системного вызова *fork()*. Он создаёт клон интерпретатора Python, включая всё состояние программы на момент копирования. В Windows запускается независимая копия интерпретатора, которая не клонирует состояние. Реального разветвления процессов не произойдёт до первого вызова метода *pool.map()* или *pool.submit()*.
- Нужно с великой осторожностью объединять пулы процессов с

программами, которые используют потоки. В частности, вы, вероятно, должны создавать и запускать пул процессов до создания любых потоков (то есть создавать пул в главном потоке на старте программы).

12.9. Разбираемся с GIL (и перестаём волноваться по этому поводу)

Задача

Вы слышали о глобальной блокировке интерпретатора (Global Interpreter Lock, GIL) и волнуетесь, что она может повлиять на производительность вашей многопоточной программы.

Решение

Хотя Python полностью поддерживает многопоточное программирование, части интерпретатора, реализованные на С, не полностью потокобезопасны и не позволяют осуществлять полностью конкурентное выполнение. На самом деле интерпретатор защищён так называемой глобальной блокировкой интерпретатора (GIL), которая позволяет только одному потоку Python выполняться в любой конкретный момент времени. Наиболее заметный эффект GIL в том, что многопоточные программы на Python не могут полностью воспользоваться преимуществами многоядерных процессоров (тяжёлые вычислительные задачи, использующие больше одного потока, работают только на одном CPU).

Перед обсуждением распространённых способов обхода GIL, важно подчеркнуть, что GIL влияет только на программы, сильно нагружающие CPU (то есть те, в которых вычисления доминируют). Если ваша программа в основном занимается вводом-выводом, что типично для сетевых коммуникаций, потоки часто являются разумным выбором, потому что они проводят большую часть времени в ожидании. На самом деле вы можете создать тысячи потоков Python без каких-либо проблем. Современные операционные системы легко справляются с запуском такого количества потоков, так что здесь вам просто не о чём волноваться.

Для написания завязанных на CPU программ вам придётся изучить природу выполняемых вычислений. Например, правильный выбор алгоритма может

дать намного более заметный прирост скорости, чем попытки распараллелить неоптимальный алгоритм на потоки. Похожим образом, принимая во внимание то, что Python является интерпретируемым языком, вы можете получить значительный прирост производительности путём выноса чувствительного к скорости кода в модуль на языке С. Расширения типа [NumPy](#) также крайне эффективны для ускорения определённых вычислений с массивами данных. И последнее: вы можете исследовать альтернативные реализации, такие как [PyPy](#), среди возможностей которой присутствует JIT-компилятор.

Также стоит отметить, что потоки используются не только для повышения производительности. Зависящие от CPU программы могут использовать потоки для управления графическим интерфейсом, сетевым соединением, или предоставлять какой-то другой сервис. В этом случае GIL может представлять серьёзную проблему, поскольку код, который удерживает её слишком долго, будет вызывать раздражающие задержки в потоках, не связанных с обработкой данных с помощью CPU. На самом деле, плохо написанное расширение на С может усугубить эту проблему, хотя часть, выполняющая вычисления, может работать быстрее.

Сказав всё это, перейдём к описанию приёмов работы. Есть две типовые стратегии обхода ограничений GIL. Во-первых, если вы работаете полностью в Python, вы можете использовать модуль *multiprocessing*, чтобы создать пул процессов и использовать его наподобие сопроцессора. Предположим, например, что у вас есть такой код на базе потоков:

```
# Выполняет тяжёлые вычисления (заявзана на CPU)
def some_work(args):
    ...
    return result

# Поток, который вызывает вышеупомянутую функцию
def some_thread():
    while True:
        ...
        r = some_work(args)
        ...


```

Вот как вы могли бы изменить код, чтобы использовать пул:

```
# Пул обработки (инициализацию см. ниже)
pool = None
```

```

# Выполняет тяжёлые вычисления (завязана на CPU)
def some_work(args):
    ...
    return result

# Поток, который вызывает вышеуказанную функцию
def some_thread():
    while True:
        ...
        r = pool.apply(some_work, (args))
        ...

# Инициализация пула
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()

```

Этот пример с пулом обходит GIL, используя хитрый трюк. Когда поток хочет выполнить тяжёлые вычисления, он передаёт работу пулу. Пул, в свою очередь, передаёт работуциальному интерпретатору Python, работающему в другом процессе. Пока поток ожидает результата, он освобождает GIL. Более того, по причине выполнения вычислений в отдельном интерпретаторе, они более не связаны ограничениями GIL. На многоядерной системе вы обнаружите, что этот приём без проблем позволяет вам выжать максимум из всех CPU.

Вторая стратегия для обхода GIL — сфокусироваться на программировании расширений на С. Общая идея в том, чтобы переместить тяжёлые вычислительные задачи в модули С, работающие независимо от Python, и заставить код на С освобождать блокировку, когда он работает. Это делается путём помещения в код на С специальных макросов:

```

#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}

```

Если вы используете другие инструменты для доступа к С, такие как библиотека *cTypes* или *Cython*, вам, возможно, не придётся ничего делать.

Например, `c_types` по умолчанию освобождает GIL при вызове кода на C.

Обсуждение

Многие программисты при встрече с проблемами производительности потоков обвиняют во всём GIL. Однако это близоруко и наивно. Приведём пример из реального мира: загадочные «зависания» в многопоточной сетевой программе могут быть вызваны совершенно другой причиной (например, подвисшим обращением к DNS), а не чем-то связанным с GIL. Короче говоря, вам нужно по-настоящему изучить свой код, чтобы понять, связана ли проблема с GIL. Повторимся: осознайте, что проблемы с GIL возникают в задачах, связанных с тяжелой обработкой на CPU, а не при работе с вводом-выводом.

Если вы будете использовать пул процессов в качестве обходного пути, то вы должны знать, что вам придётся применять сериализацию данных и коммуникацию с другим интерпретатором Python. Чтобы это работало, операция, которая должна быть выполнена, должна быть упакована в функцию, определённую через инструкцию `def` (то есть лямбды, замыкания, вызываемые объекты не подойдут) и аргументы функции, а возвращаемое значение должно быть совместимо с `pickle`. Также объем выполняемой работы должен быть достаточно большим, чтобы перекрыть оверхед на дополнительную коммуникацию.

Другой тонкий аспект в том, что смешивание пулов процессов и пулов потоков — отличный способ взорвать себе голову. Если вы используете эти возможности вместе, часто лучше создать пул процессов как синглтон (паттерн «Одиночка») при запуске программы, перед созданием потоков. Потоки затем используют один и тот же пул процессов для всей тяжелой вычислительной работы.

Для расширений на языке C очень важно поддерживать изоляцию от процесса интерпретатора Python. Если вы хотите переложить работу с Python на C, то убедитесь, что код на C работает независимо от Python. Это означает, что не надо использовать структуры данных Python, и не надо вызывать API Python для работы с C. Ещё один момент — убедитесь, что расширение на C выполняет достаточный объем работы, чтобы оправдать все усилия по его подключению. Намного лучше, когда расширение выполняет миллионы вычислений, нежели несколько небольших.

Очевидно, что эти решения для обхода GIL не универсальны и не подходят для абсолютно любой задачи. Например, некоторые типы приложений не очень хорошо работают при разделении на несколько процессов или при попытках переписать часть кода на C. Для таких приложений вы можете придумать собственное решение (например, несколько процессов, осуществляющих доступ к общим областям памяти, несколько интерпретаторов, работающих в одном процессе и т.д.) В качестве альтернативы вы можете взглянуть на другие реализации интерпретатора, такие как PyPy.

См. рецепт 15.7. и рецепт 15.10., где приведена дополнительная информация об освобождении GIL в расширениях на C.

12.10. Определение акторной задачи

Задача

Вы хотите определять задачи с поведением, аналогичным «акторам» в так называемой «модели акторов».

Решение

«Модель акторов» — один из самых старых и простых подходов к конкурентности и распределённым вычислениям. На самом деле, простота — это часть её привлекательности. Если вкратце, то актор — это конкурентно выполняемая задача, которая просто производит действия над посыпаемыми ей сообщениями. В ответ на эти сообщения актор может решить послать другие сообщения другим акторам. Коммуникация между акторами односторонняя и асинхронная. Отправляющий сообщение не знает, когда сообщение будет доставлено, а также не получает ответа или уведомления об обработке сообщения.

Акторы определяются без особых хитростей — путём объединения потока и очереди. Например:

```
from queue import Queue
from threading import Thread, Event

# Страж, использующийся для отключения
class ActorExit(Exception):
```

```
pass

class Actor:
    def __init__(self):
        self._mailbox = Queue()

    def send(self, msg):
        """
        Посыпает сообщение в актор
        """
        self._mailbox.put(msg)

    def recv(self):
        """
        Получает входящее сообщение
        """
        msg = self._mailbox.get()
        if msg is ActorExit:
            raise ActorExit()
        return msg

    def close(self):
        """
        Закрывает актор и отключает его
        """
        self.send(ActorExit)

    def start(self):
        """
        Запускает конкурентное выполнение
        """
        self._terminated = Event()
        t = Thread(target=self._bootstrap)
        t.daemon = True
        t.start()

    def _bootstrap(self):
        try:
            self.run()
        except ActorExit:
            pass
        finally:
            self._terminated.set()

    def join(self):
        self._terminated.wait()

    def run(self):
        """
        Запускает метод, который реализует пользователь
        """

```

```

while True:
    msg = self.recv()

# Пример ActorTask
class PrintActor(Actor):
    def run(self):
        while True:
            msg = self.recv()
            print('Got:', msg)

# Пример использования
p = PrintActor()
p.start()
p.send('Hello')
p.send('World')
p.close()
p.join()

```

В этом примере экземпляры *Actor* — это штуки, которые просто посылают сообщение, используя свой метод *send()*. «Под капотом» тут происходит помещение сообщения в очередь и передача его внутреннему потоку, который запущен для обработки полученных сообщений. Метод *close()* запрограммирован, чтобы отключать актор путём помещения специального «сторожевого» значения (*ActorExit*) в очередь. Пользователи определяют новые акторы путём наследования от *Actor* и переопределения метода *run()* для реализации кастомной обработки. Особенность исключения *ActorExit* в том, что определённый пользователем код может быть запрограммирован так, чтобы поймать запрос на завершение и правильно его обработать (исключение возбуждается методом *get()* и распространяется).

Если вы ослабите требования к конкурентности и асинхронной доставке сообщений, актороподобные объекты могут быть минималистично определены с помощью генераторов. Например:

```

def print_actor():
    while True:
        try:
            msg = yield      # Получаем сообщение
            print('Got:', msg)
        except GeneratorExit:
            print('Actor terminating')

# Пример использования
p = print_actor()
next(p)                  # Продвигаемся к yield (готовы к получению)
p.send('Hello')

```

```
p.send('World')
p.close()
```

Обсуждение

Важная часть привлекательности акторов — их простота. На практике у нас есть только одна основная операция — `send()`. А общая концепция «сообщения» в основанных на акторах системах может быть расширена по многим различным направлениям. Например, вы могли бы передавать помеченные тегами сообщения в форме кортежей и заставлять акторы принимать различные решения:

```
class TaggedActor(Actor):
    def run(self):
        while True:
            tag, *payload = self.recv()
            getattr(self, 'do_'+tag)(*payload)

    # Методы, соответствующие различным тегам сообщений
    def do_A(self, x):
        print('Running A', x)

    def do_B(self, x, y):
        print('Running B', x, y)

    # Пример
    a = TaggedActor()
    a.start()
    a.send(('A', 1))           # Вызывает do_A(1)
    a.send(('B', 2, 3))         # Вызывает do_B(2,3)
```

А вот другой пример — актор, который позволяет произвольной функции выполняться в воркере, а результатам — отправляться обратно с использованием специального объекта `Result`:

```
from threading import Event
class Result:
    def __init__(self):
        self._evt = Event()
        self._result = None

    def set_result(self, value):
        self._result = value
        self._evt.set()

    def result(self):
```

```
    self._evt.wait()
    return self._result

class Worker(Actor):
    def submit(self, func, *args, **kwargs):
        r = Result()
        self.send((func, args, kwargs, r))
        return r

    def run(self):
        while True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))

# Пример использования
worker = Worker()
worker.start()
r = worker.submit(pow, 2, 3)
print(r.result())
```

И последнее: концепция «отсылки» задачи в сообщении может быть масштабирована в системы, которые используют несколько процессов, или даже в крупные распределённые системы. Например, метод `send()` актороподобных объектов может быть запрограммирован для передачи данных по соединению через сокет или доставки данных через какую-то инфраструктуру передачи сообщений (например, AMQP, ZMQ и т.д.)

12.11. Реализация системы сообщений «опубликовать/подписаться» (pub/sub)

Задача

У вас есть программа на базе коммуницирующих потоков, и вы хотите реализовать систему сообщений «опубликовать/подписаться».

Решение

Чтобы реализовать систему сообщений «опубликовать/подписаться», обычно вам нужно добавить отдельный объект «пункт обмена» или «шлюз», который действует как посредник для всех сообщений. Вместо прямой отсылки сообщений из одной задачи в другую, сообщение посыпается в пункт обмена, а он доставляет сообщение в одну или более прикреплённых задач. Вот

пример очень простой реализации пункта обмена:

```
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Словарь всех созданных пунктов обмена
_exchanges = defaultdict(Exchange)

# Вернуть экземпляр Exchange, ассоциированный с переданным именем
def get_exchange(name):
    return _exchanges[name]
```

Пункт обмена — это просто объект, который хранит множество активных подписчиков и предоставляет методы для прикрепления, открепления и отсылки сообщений. Каждый пункт обмена идентифицируется по имени, а функция `get_exchange()` просто возвращает экземпляр `Exchange`, связанный с переданным именем.

Вот простой пример, который демонстрирует использование пункта обмена:

```
# Пример задачи. Любой объект с методом send()

class Task:
    ...
    def send(self, msg):
        ...

task_a = Task()
task_b = Task()

# Пример получения пункта обмена
exc = get_exchange('name')

# Пример подписывания задач на него
exc.attach(task_a)
```

```
exc.attach(task_b)

# Пример отсылки сообщений
exc.send('msg1')
exc.send('msg2')

# Пример отписки
exc.detach(task_a)
exc.detach(task_b)
```

Хотя есть много различных вариантов этой темы, общая идея остаётся одинаковой. Сообщения будут доставлены в пункт обмена, а пункт обмена доставит их прикреплённым подписчикам.

Обсуждение

Концепция задач или потоков, посылающих друг другу сообщения (часто через очереди), легко реализуется и весьма популярна. Однако преимущества использования вместо неё модели «опубликовать/подписаться» (pub/sub) часто упускаются из вида.

Во-первых, использование пункта обмена может упростить большую часть грязных сантехнических работ, с которыми нужно возиться при настройке общающихся потоков. Вместо того, чтобы пытаться связать вместе потоки по множеству модулей программы, вам нужно только позаботиться о подсоединении их к известному пункту обмена. В каком-то смысле это похоже на работу библиотеки *logging*. На практике это облегчает разделение различных задач в программе.

Во-вторых, способность пункта обмена «широковещательно транслировать» сообщения множеству подписчиков открывает возможности для создания новых паттернов коммуникации. Например, вы могли бы реализовать системы с избыточностью задач, широковещательные системы или системы с большим количеством соединений между объектами. Вы также можете построить инструменты для отладки и диагностики, которые прикрепляют себя к пункту обмена как обычные подписчики. Например, вот простой диагностический класс, который мог бы показывать отосланные сообщения:

```
class DisplayMessages:
    def __init__(self):
        self.count = 0
    def send(self, msg):
        self.count += 1
```

```
print('msg[{}]: {!r}'.format(self.count, msg))

exc = get_exchange('name')
d = DisplayMessages()
exc.attach(d)
```

И последний важный аспект этой реализации: она работает с различными «задачеподобными» объектами. Например, получатели сообщения могут быть акторами (как описано в [рецепте 12.10.](#)), корутинами (сопрограммами), сетевыми соединениями или просто чем угодно, что реализует правильный метод `send()`.

Потенциальная проблемная сторона пункта сообщений — это правильное прикрепление и открепление подписчиков. Чтобы правильно управлять ресурсами, каждый прикреплённый подписчик рано или поздно должен быть откреплён. Это ведёт к модели программирования, похожей на такую:

```
exc = get_exchange('name')
exc.attach(some_task)
try:
    ...
finally:
    exc.detach(some_task)
```

В каком-то смысле это похоже на использование файлов, блокировок и других подобных объектов. Опыт показывает, что очень легко забыть о финальном шаге `detach()`. Чтобы упростить это, вам стоит подумать об использовании протокола менеджера контекста. Например, добавить метод `subscribe()` в пункт обмена таким образом:

```
from contextlib import contextmanager
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    @contextmanager
    def subscribe(self, *tasks):
```

```

        for task in tasks:
            self.attach(task)
    try:
        yield
    finally:
        for task in tasks:
            self.detach(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Словарь всех созданных пунктов обмена
_exchanges = defaultdict(Exchange)

# Возвращает экземпляр Exchange, ассоциированный с переданным именем
def get_exchange(name):
    return _exchanges[name]

# Пример использования метода subscribe()
exc = get_exchange('name')
with exc.subscribe(task_a, task_b):
    ...
    exc.send('msg1')
    exc.send('msg2')
    ...

# task_a и task_b открепляются здесь

```

В конце стоит отметить, что у идеи пункта обмена есть огромное количество возможностей для расширения. Например, пункты обмена могут реализовывать весь набор каналов обмена сообщениями или применять правила паттерн-матчинга (сопоставления с образцом) к именам пунктов обмена. Пункты обмена также могут быть расширены до приложений распределённых вычислений (например, переправки сообщений задачам на разных компьютерах и т.п.)

12.12. Использование генераторов в качестве альтернативы потокам

Задача

Вы хотите реализовать конкурентность, используя генераторы (корутины) в качестве альтернативы системным потокам. Такой подход иногда называют

потоками на уровне пользователя или зелёными потоками.

Решение

Чтобы реализовать собственную конкурентность, используя генераторы, сначала вам нужно осознать фундаментальную идею функций-генераторов и инструкции *yield*. Если точнее, основа поведения *yield* — это то, что она заставляет генератор приостановить своё выполнение. Используя приостановку выполнения, можно написать планировщик, который обращается с генераторами, как со своего рода «задачами», и изменять их выполнение, используя некое кооперативное переключение задач.

Чтобы проиллюстрировать эту идею, рассмотрите две функции-генератора, использующих простой *yield*:

```
# Два простых генератора
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1
    print('Blastoff!')

def countup(n):
    x = 0
    while x < n:
        print('Counting up', x)
        yield
        x += 1
```

Эти функции, возможно, выглядят немного странно, потому что используют *yield* сами по себе. Однако рассмотрим следующий код, который реализует простой планировщик задач:

```
from collections import deque

class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        ...
        Допускает новую запущенную задачу в планировщик
        ...
        self._task_queue.append(task)
```

```

def run(self):
    ...
    Работает, пока не останется задач
    ...

    while self._task_queue:
        task = self._task_queue.popleft()
        try:
            # Работаем до следующей инструкции yield
            next(task)
            self._task_queue.append(task)
        except StopIteration:
            # Генератор более не выполняется
            pass

# Пример использования
sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()

```

В этом коде класс *TaskScheduler* запускает коллекцию генераторов в стиле round-robin — каждый работает, пока они не достигнут инструкции *yield*. Например, вывод будет таким:

```

T-minus 10
T-minus 5
Counting up 0
T-minus 9
T-minus 4
Counting up 1
T-minus 8
T-minus 3
Counting up 2
T-minus 7
T-minus 2
...

```

В этой точке вы, по сути, реализовали маленькое ядро «операционной системы». Функции-генераторы — это задачи, а инструкция *yield* — это способ, с помощью которого задачи сигнализируют о том, что хотят приостановиться. Планировщик просто проходит по задачам в цикле, пока не останется ни одной выполняющейся.

На практике вы, вероятно, не будете использовать генераторы для реализации конкурентности в такой простой форме, как показано выше.

Вместо этого вы можете использовать генераторы, чтобы заменить потоки при реализации акторов (см. рецепт 12.10.) или сетевые серверы.

Следующий код иллюстрирует использование генераторов для реализации акторов без использования потоков:

```
from collections import deque

class ActorScheduler:
    def __init__(self):
        self._actors = {}           # Отображение имён на акторы
        self._msg_queue = deque()   # Очередь сообщений

    def new_actor(self, name, actor):
        """
        Допускает новый запущенный актор в планировщик и даёт ему имя
        """
        self._msg_queue.append((actor, None))
        self._actors[name] = actor

    def send(self, name, msg):
        """
        Посыпает сообщение актору с соответствующим именем
        """
        actor = self._actors.get(name)
        if actor:
            self._msg_queue.append((actor, msg))

    def run(self):
        """
        Работает до тех пор, пока в очереди есть сообщения
        """
        while self._msg_queue:
            actor, msg = self._msg_queue.popleft()
            try:
                actor.send(msg)
            except StopIteration:
                pass

    # Пример использования
if __name__ == '__main__':
    def printer():
        while True:
            msg = yield
            print('Got:', msg)

    def counter(sched):
        while True:
            # Получить текущий счёт
            n = yield
```

```

    if n == 0:
        break
    # Послать задаче-принтеру
    sched.send('printer', n)
    # Послать следующий счёт задаче-счётчику (рекурсивно)
    sched.send('counter', n-1)

sched = ActorScheduler()
# Создать первоначальные акторы
sched.new_actor('printer', printer())
sched.new_actor('counter', counter(sched))

# Послать начальное сообщение в счётчик для инициализации
sched.send('counter', 10000)
sched.run()

```

Для понимания этого кода нужно попытаться, но ключевой момент тут — очередь ожидающих сообщений. По сути, планировщик работает до тех пор, пока есть сообщения, которые нужно доставить. Стоит отметить, что генератор *counter* отсылает сообщения себе и работает в рекурсивном цикле, не ограниченном лимитом на рекурсию Python.

Вот продвинутый пример, показывающий использование генераторов для реализации конкурентного сетевого приложения:

```

from collections import deque
from select import select

# Этот класс представляет общее yield-событие в планировщике
class YieldEvent:
    def handle_yield(self, sched, task):
        pass
    def handle_resume(self, sched, task):
        pass

# Планировщик задач
class Scheduler:
    def __init__(self):
        self._numtasks = 0          # Общее количество задач
        self._ready = deque()       # Задачи, готовые к запуску
        self._read_waiting = {}     # Задачи, ждущие чтения
        self._write_waiting = {}    # Задачи, ждущие записи

    # Опрашивает на события ввода-вывода и перезапускает ждущие задачи
    def _iopoll(self):
        rset,wset,eset = select(self._read_waiting,
                               self._write_waiting,[])
        for r in rset:

```

```

        evt, task = self._read_waiting.pop(r)
        evt.handle_resume(self, task)
    for w in wset:
        evt, task = self._write_waiting.pop(w)
        evt.handle_resume(self, task)

def new(self,task):
    """
    Добавляет новую запущенную задачу в планировщик
    """
    self._ready.append((task, None))
    self._numtasks += 1

def add_ready(self, task, msg=None):
    """
    Добавляет уже запущенную задачу в очередь готовых.
    msg – это то, что посыпается в задачу, когда она
    возобновляется.
    """
    self._ready.append((task, msg))

# Добавляет задачу в множество чтения
def _read_wait(self, fileno, evt, task):
    self._read_waiting[fileno] = (evt, task)

# Добавляет задачу в множество записи
def _write_wait(self, fileno, evt, task):
    self._write_waiting[fileno] = (evt, task)

def run(self):
    """
    Запускает планировщик задач, пока задач не останется
    """
    while self._numtasks:
        if not self._ready:
            self._iopoll()
        task, msg = self._ready.popleft()
        try:
            # Запустить корутину к следующему yield
            r = task.send(msg)
            if isinstance(r, YieldEvent):
                r.handle_yield(self, task)
            else:
                raise RuntimeError('unrecognized yield event')
        except StopIteration:
            self._numtasks -= 1

# Пример реализации сокетного ввода-вывода на основе корутин
class ReadSocket(YieldEvent):
    def __init__(self, sock, nbytes):

```

```

        self.sock = sock
        self.nbytes = nbytes
    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        data = self.sock.recv(self.nbytes)
        sched.add_ready(task, data)

class WriteSocket(YieldEvent):
    def __init__(self, sock, data):
        self.sock = sock
        self.data = data
    def handle_yield(self, sched, task):
        sched._write_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        nsent = self.sock.send(self.data)
        sched.add_ready(task, nsent)

class AcceptSocket(YieldEvent):
    def __init__(self, sock):
        self.sock = sock
    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        r = self.sock.accept()
        sched.add_ready(task, r)

# Обёртка вокруг объекта сокета для использования с yield
class Socket(object):
    def __init__(self, sock):
        self._sock = sock
    def recv(self, maxbytes):
        return ReadSocket(self._sock, maxbytes)
    def send(self, data):
        return WriteSocket(self._sock, data)
    def accept(self):
        return AcceptSocket(self._sock)
    def __getattr__(self, name):
        return getattr(self._sock, name)

if __name__ == '__main__':
    from socket import socket, AF_INET, SOCK_STREAM
    import time

    # Пример функции, использующей генераторы. Это нужно вызывать
    # с использованием line = yield from readline(sock)
    def readline(sock):
        chars = []
        while True:
            c = yield sock.recv(1)
            if not c:

```

```

        break
    chars.append(c)
    if c == b'\n':
        break
return b''.join(chars)

# Эхо-сервер, использующий генераторы
class EchoServer:
    def __init__(self,addr,sched):
        self.sched = sched
        sched.new(self.server_loop(addr))

    def server_loop(self,addr):
        s = Socket(socket(AF_INET,SOCK_STREAM))
        s.bind(addr)
        s.listen(5)
        while True:
            c,a = yield s.accept()
            print('Got connection from ', a)
            self.sched.new(self.client_handler(Socket(c)))

    def client_handler(self,client):
        while True:
            line = yield from readline(client)
            if not line:
                break
            line = b'GOT:' + line
            while line:
                nsent = yield client.send(line)
                line = line[nsent:]
        client.close()
        print('Client closed')

sched = Scheduler()
EchoServer(' ',16000),sched)
sched.run()

```

Этот код, несомненно, требует времени для внимательного изучения. Однако это, по сути, реализация маленькой операционной системы. Здесь есть очередь задач, готовых к запуску, а также ожидающие области для задач, спящих в ожидании ввода-вывода. Большая часть планировщика занимается перемещением задач между очередью готовых и ожидающей ввода-вывода областью.

Обсуждение

При создании фреймворка на базе генераторов, обеспечивающих

конкурентность, наиболее часто работают с более общей формой *yield*:

```
def some_generator():
    ...
    result = yield data
    ...
```

Функции, которые используют *yield* этим способом, часто называются «корутинами» (сопрограммами). Внутри планировщика инструкция *yield* обрабатывается в цикле:

```
f = some_generator()

# Первоначальный результат. Это None на старте,
# поскольку ничего ещё не вычислено
result = None
while True:
    try:
        data = f.send(result)
        result = ... do some calculation ...
    except StopIteration:
        break
```

Обсуждение

Логика, касающаяся *result*, достаточно извилиста. Передаваемое в *send()* значение определяет, что будет возвращено, когда инструкция *yield* проснётся. Итак, если *yield* вернёт результат в ответ на данные, которые были ранее выданы (*yielded*), он будет возвращён в следующей операции *send()*. Если функция-генератор только что стартовала, посылка есть значения *None* просто заставляет продвинуться к первой инструкции *yield*.

В дополнение к отсылке значений, также можно выполнить метод *close()* генератора. Это приведет к возбуждению тихого исключения *GeneratorExit* при достижении инструкции *yield*, что остановит выполнение. Если вы пожелаете, генератор может поймать это исключение и выполнить действия по очистке. Также можно использовать метод генератора *throw()*, чтобы возбудить произвольное исключение в инструкции *yield*. Планировщик задач может использовать это для передачи сообщений об ошибках в запущенных генераторах.

Инструкция *yield from*, использованная в последнем примере, применяется

для реализации корутин, которые служат в качестве подпрограмм или процедур, вызываемых из других генераторов. По сути, поток управления прозрачно переходит в новую функцию. В отличие от обычных генераторов, функции, которые вызываются через *yield from*, могут возвращать значение, которое становится результатом выполнения инструкции *yield from*.

Дополнительные сведения об этой инструкции вы найдете в [PEP 380](#).

И последнее: при программировании с использованием генераторов важно знать о нескольких базовых ограничениях. В частности, вы не получите преимуществ, предоставляемых потоками. Например, если вы выполняете любой завязанный на CPU код (или код, блокирующий ввод-вывод), он приостановит весь планировщик задач до завершения операции.

Единственная возможность обойти это — делегировать операциюциальному потоку или процессу, где она будет работать независимо. Ещё одно ограничение в том, что большинство библиотек Python не писались с целью хорошо работать с потоками на базе генераторов. Если вы примените этот подход, то обнаружите, что вам нужно писать замены для многих стандартных библиотечных функций.

В качестве базового материала о корутинах и использованных в этом рецепте приёмах рекомендуем [PEP 342](#) и “[A Curious Course on Coroutines and Concurrency](#)”.

[PEP 3156](#) описывает современный подход к асинхронному вводу-выводу с использованием корутин. На практике вы очень вряд ли будете писать низкоуровневый планировщик корутин. Однако идеи, окружающие корутины, являются базой для многих популярных библиотек: [gevent](#), [greenlet](#), [Stackless Python](#) и других похожих проектов.

12.13. Опрашивание многопоточных очередей

Задача

У вас есть коллекция потоковых очередей, и вы хотите опрашивать их на предмет входящих элементов — примерно так же, как вы могли бы опрашивать коллекцию сетевых соединений на предмет входящих данных.

Решение

Обычное решение задачи опрашивания подразумевает использование малоизвестного трюка со скрытым зацикленным сетевым соединением. По сути, идея такова: для каждой очереди (или любого объекта), которую вы хотите опрашивать, вы создаете пару соединённых сокетов. Затем вы пишете в один из сокетов, чтобы просигнализировать о присутствии данных. Другой сокет затем передаётся в `select()` или похожую функцию для опрашивания на предмет прибытия данных. Вот пример кода, иллюстрирующий эту идею:

```
import queue
import socket
import os

class PollableQueue(queue.Queue):
    def __init__(self):
        super().__init__()
        # Создаём пару соединённых сокетов
        if os.name == 'posix':
            self._putsocket, self._getsocket = socket.socketpair()
        else:
            # Для совместимости с не-POSIX-системами
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self._putsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self._putsocket.connect(server.getsockname())
            self._getsocket, _ = server.accept()
            server.close()

    def fileno(self):
        return self._getsocket.fileno()

    def put(self, item):
        super().put(item)
        self._putsocket.send(b'x')

    def get(self):
        self._getsocket.recv(1)
        return super().get()
```

Здесь мы определили экземпляр `Queue` нового типа, в котором зашита пара соединённых сокетов. Функция `socketpair()`, доступная на компьютерах с Unix, может легко установить такие сокеты. На Windows вам придётся подделать её, используя код, похожий на показанный (он выглядит немного странно, но

после создания серверного сокета клиент немедленно подключается к нему). Обычные методы `get()` и `put()` здесь слегка переопределены, чтобы выполнять «минимальный» ввод-вывод на этих сокетах. Метод `put()` пишет единичный байт данных в один из сокетов — после помещения данных в очередь. Метод `get()` читает единичный байт из другого сокета и затем удаляет элемент из очереди.

Метод `fileno()` — это то, что позволяет опрашивать очередь, используя методы типа `select()`. По сути, он просто показывает внутренний файловый дескриптор сокета, используемого функцией `get()`.

Вот пример кода, который определяет консьюмера (потребителя), отслеживающего появление элементов во многих очередях сразу:

```
import select
import threading

def consumer(queues):
    ...

    Консьюмер, который одновременно читает данные из нескольких очередей
    ...

    while True:
        can_read, _, _ = select.select(queues, [], [])
        for r in can_read:
            item = r.get()
            print('Got:', item)

q1 = PollableQueue()
q2 = PollableQueue()
q3 = PollableQueue()
t = threading.Thread(target=consumer, args=(q1,q2,q3,))
t.daemon = True
t.start()

# Скармливаем данные очередям
q1.put(1)
q2.put(10)
q3.put('hello')
q2.put(15)
...
```

Если вы попробуете поэкспериментировать, то обнаружите, что консьюмер на самом деле получает все помещаемые в очередь элементы — независимо от того, в какую из очередей они помещаются.

Обсуждение

Задача опрашивания нефайлоподобных объектов, таких как очереди, часто сложнее, чем кажется. Например, если вы не используете показанный приём с сокетами, ваш единственный вариант — написать код, который в цикле обходит очереди и использует таймер:

```
import time

def consumer(queues):
    while True:
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)

        # Ненадолго засыпаем, чтобы избежать 100%-ной загрузки CPU
        time.sleep(0.01)
```

Для некоторых задач это может подойти, но приём довольно неуклюжий и привносит странные проблемы с производительностью. Например, если новые данные добавлены в очередь, они не будут обнаружены в течение 10 миллисекунд (для современных процессоров это целая вечность).

Вы вляпаетесь в ещё более серьёзные проблемы, если показанное выше опрашивание смешано с опрашиванием других объектов, таких как сетевые сокеты. Например, если вы хотите опрашивать и сокеты, и очереди одновременно, вы должны будете использовать такой код:

```
import select

def event_loop(sockets, queues):
    while True:
        # Опрашивание с таймаутом
        can_read, _, _ = select.select(sockets, [], [], 0.01)
        for r in can_read:
            handle_read(r)
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)
```

Показанное решение справляется с большинством этих проблем, просто помещая очереди на одну ступеньку с сокетами. Единственный вызов `select()`

может быть использован для опрашивания наличия активности у обоих. Более того, если данные помещаются в очередь, консьюмер будет уведомлен практически одновременно. Хотя здесь присутствует небольшой оверхед на «подкапотный» ввод-вывод, часто это того стоит, поскольку предлагает лучшее время отклика и упрощает код.

12.14. Запуск процесса-демона на Unix

Задача

Вы хотели бы написать программу, которая работает, как настоящий процесс-демон на Unix или Unix-подобных системах.

Решение

Создание правильного процесса-демона требует точного соблюдения последовательности системных вызовов и внимания к деталям. Приведённый ниже код показывает, как определить процесс-демон, который можно легко остановить после запуска:

```
#!/usr/bin/env python3
# daemon.py

import os
import sys
import atexit
import signal

def daemonize(pidfile, *, stdin='/dev/null',
              stdout='/dev/null',
              stderr='/dev/null'):

    if os.path.exists(pidfile):
        raise RuntimeError('Already running')

    # Первый форк (отделяется от родителя)
    try:
        if os.fork() > 0:
            raise SystemExit(0)      # Parent exit
    except OSError as e:
        raise RuntimeError('fork #1 failed.')

    os.chdir('/')
    os.umask(0)
```

```
os.setsid()
# Второй форк (уступает лидерство в сессии)
try:
    if os.fork() > 0:
        raise SystemExit(0)
except OSError as e:
    raise RuntimeError('fork #2 failed.')

# Сброс буферов ввода-вывода
sys.stdout.flush()
sys.stderr.flush()

# Заменяет файловые дескрипторы для stdin, stdout и stderr
with open(stdin, 'rb', 0) as f:
    os.dup2(f.fileno(), sys.stdin.fileno())
with open(stdout, 'ab', 0) as f:
    os.dup2(f.fileno(), sys.stdout.fileno())
with open(stderr, 'ab', 0) as f:
    os.dup2(f.fileno(), sys.stderr.fileno())

# Записывает PID-файл
with open(pidfile, 'w') as f:
    print(os.getpid(), file=f)

# Устраиваем так, чтобы удалить PID-файл по выходу/сигналу
atexit.register(lambda: os.remove(pidfile))

# Обработчик сигнала для завершения (требуется)
def sigterm_handler(signo, frame):
    raise SystemExit(1)

signal.signal(signal.SIGTERM, sigterm_handler)

def main():
    import time
    sys.stdout.write('Daemon started with pid {}\n'.format(os.getpid()))
    while True:
        sys.stdout.write('Daemon Alive! {}\n'.format(time.ctime()))
        time.sleep(10)

if __name__ == '__main__':
    PIDFILE = '/tmp/daemon.pid'

    if len(sys.argv) != 2:
        print('Usage: {} [start|stop]'.format(sys.argv[0]), file=sys.stderr)
        raise SystemExit(1)

    if sys.argv[1] == 'start':
        try:
            daemonize(PIDFILE,
                      stdout='/tmp/daemon.log',
```

```
        stderr='/tmp/dameon.log')
    except RuntimeError as e:
        print(e, file=sys.stderr)
        raise SystemExit(1)

main()

elif sys.argv[1] == 'stop':
    if os.path.exists(PIDFILE):
        with open(PIDFILE) as f:
            os.kill(int(f.read()), signal.SIGTERM)
    else:
        print('Not running', file=sys.stderr)
        raise SystemExit(1)
else:
    print('Unknown command {!r}'.format(sys.argv[1]), file=sys.stderr)
    raise SystemExit(1)
```

Чтобы запустить демон, пользователь должен использовать такую команду:

```
bash % daemon.py start
bash % cat /tmp/daemon.pid
2882
bash % tail -f /tmp/daemon.log
Daemon started with pid 2882
Daemon Alive! Fri Oct 12 13:45:37 2012
Daemon Alive! Fri Oct 12 13:45:47 2012
...
...
```

Процессы-демоны работают в бэкграунде, так что команда возвращается сразу же. Однако вы можете просмотреть его pid-файл и лог, как показано выше. Чтобы остановить демон, напишите:

```
bash % daemon.py stop
bash %
```

Обсуждение

В этом рецепте мы определяем функцию *daemonize()*, которая должна быть вызвана при запуске программы, чтобы заставить программу выполняться в качестве демона. Сигнатура *daemonize()* использует обязательные именованные аргументы, чтобы сделать назначение необязательных аргументов более ясным. Это заставляет пользователя использовать такие вызовы:

```
daemonize('daemon.pid',
          stdin='/dev/null',
          stdout='/tmp/daemon.log',
          stderr='/tmp/daemon.log')
```

Такой вызов был бы более загадочным:

```
# Недопустимо. Необходимо использовать именованные аргументы
daemonize('daemon.pid',
           '/dev/null', '/tmp/daemon.log', '/tmp/daemon.log')
```

Шаги создания демона достаточно малопонятны, но общая идея такова: во-первых, демон должен отделить себя от своего родительского процесса. Это цель первой операции `os.fork()` и немедленного завершения родителя.

После того, как потомок осиротел, вызов `os.setsid()` создаёт полностью новую сессию процесса и устанавливает потомка лидером. Это также устанавливает потомка лидером новой группы процессов и позволяет убедиться, что отсутствует контролирующий терминал. Если всё это звучит для вас как магические заклинания, то просто знайте, что это связано с необходимостью правильно отделить демона от терминала и убедиться, что такие штуки, как сигналы, не мешают его работе.

Вызовы `os.chdir()` и `os.umask()` меняют текущий рабочий каталог и сбрасывают маску режима файла. Изменить каталог — это обычно хорошая идея, поскольку демон не работает в каталоге, в котором он был запущен.

Второй вызов `os.fork()` — намного более загадочная операция. Этот шаг заставляет процесс-демон отказаться от возможности получать новый контролирующий терминал и предоставляет даже большую изоляцию (по сути, демон отказывается от лидерства в сессии и более не имеет возможности открывать контролирующие терминалы). Хотя вы могли бы пропустить этот шаг, обычно рекомендуется этого не делать.

Когда демон правильно отделён, он выполняет шаги, чтобы переинициализировать стандартные потоки ввода-вывода, чтобы они указывали на файлы, определённые пользователем. Эта часть достаточно сложна. Ссылки на файловые объекты, ассоциированные со стандартными потоками ввода-вывода, находятся во многих местах интерпретатора (`sys.stdout`, `sys.__stdout__` и т.д.) Простое закрытие `sys.stdout` и переназначение вряд ли будет работать правильно, потому что нет способа

узнать, поправили ли мы все места, где используется `sys.stdout`. Вместо этого открывается отдельный файловый объект, и вызов `os.dup2()` используется, чтобы заменить им файловый дескриптор, в настоящий момент используемый `sys.stdout`. Когда это произойдёт, изначальный файл, связанный с `sys.stdout`, будет закрыт, а новый займёт его место. Нужно подчеркнуть, что любая кодировка файла или обработка текста, уже применённые к стандартным потокам ввода-вывода, останутся на месте.

Обычная вещь при работе с демонами — записывать ID процесса демона в файл, чтобы позже использовать его в других программах. Последняя часть функции `daemonize()` записывает этот файл, но также готовит всё для удаления этого файла после завершения программы. Функция `atexit.register()` регистрирует функцию для выполнения при завершении интерпретатора Python. Определение обработчика сигнала для `SIGTERM` также требуется для аккуратного завершения. Обработчик сигнала всего лишь возбуждает `SystemExit()`, не более того. Это может выглядеть ненужным, но без этого сигнал на прекращение убьёт интерпретатор без выполнения действий по очистке, зарегистрированных с помощью `atexit.register()`. Пример кода, который убивает демона, может быть найден в обработчике команды `stop` в конце программы.

Больше сведений о написании демонов вы можете найти во втором издании *Advanced Programming in the UNIX Environment* (У. Ричард Стивенс и Стивен Э. Раго, Wesley, 2005). Хотя в книге обсуждается программирование на C, материал легко адаптировать к Python, поскольку все нужные POSIX-функции доступны в стандартной библиотеке.

13. Полезные скрипты и системное администрирование

Очень многие используют Python для замены скриптов командной оболочки и автоматизации обычных системных задач. Главная цель этой главы — описать возможности, связанные с обычными задачами, которые полезны при написании скриптов. Например, парсинг аргументов командной строки, манипуляции с файлами в файловой системе, получение информации о конфигурации системы и т.д. Общая информация о файлах и каталогах также есть в [главе 5](#).

13.1. Скрипты, принимающие ввод через перенаправление, каналы или файлы

Задача

Вы хотите, чтобы скрипт, который вы пишете, мог принимать ввод, используя тот механизм, который удобен пользователю. Список механизмов должен включать передачу скрипту вывода команды по каналу (пайпу), перенаправление файла в скрипт, или же просто передачу имени файла или списка имён файлов скрипту в командной строке.

Решение

Встроенный модуль *fileinput* даёт простое и короткое решение. Если у вас есть такой скрипт:

```
#!/usr/bin/env python3
import fileinput

with fileinput.input() as f_input:
    for line in f_input:
        print(line, end='')
```

...то вы уже можете принимать ввод в скрипт всеми путями, описанными выше. Если вы сохраните скрипт под именем *filein.py* и дадите ему права на выполнение, вы можете сделать всё нижеперечисленное и получить ожидаемый вывод:

```
$ ls | ./filein.py          # Выводит содержимое каталога в stdout.
$ ./filein.py /etc/passwd   # Читает /etc/passwd в stdout.
$ ./filein.py < /etc/passwd # Читает /etc/passwd в stdout.
```

Обсуждение

Функция *fileinput.input()* создаёт и возвращает экземпляр класса *FileInput*. В дополнение к тому, что он содержит несколько удобных вспомогательных методов, экземпляр также может использоваться в качестве менеджера контекста. Итак, если мы напишем скрипт, от которого ожидается печать вывода от нескольких файлов одновременно, мы можем заставить его

включить в вывод имя файла и номер строки:

```
>>> import fileinput  
>>> with fileinput.input('/etc/passwd') as f:  
>>>     for line in f:  
...         print(f.filename(), f.lineno(), line, end=' ')  
...  
/etc/passwd 1 ##  
/etc/passwd 2 # База пользователей  
/etc/passwd 3 #  
  
<other output omitted>
```

Использование его в качестве менеджера контекста позволяет убедиться, что файл будет закрыт, если он уже не используется. Также мы можем использовать удобные методы класса *FileInput*, чтобы поместить дополнительную информацию в вывод.

13.2. Завершение программы с выводом сообщения об ошибке

Задача

Вы хотите, чтобы ваша программа завершалась, выводя сообщение в стандартный поток вывода ошибок и возвращая ненулевой код статуса.

Решение

Чтобы заставить программу завершаться таким образом, возбудите исключение *SystemExit*, но передайте ему сообщение об ошибке в качестве аргумента. Например:

```
raise SystemExit('It failed!')
```

Это приведёт к тому, что предоставленное сообщение будет выведено в *sys.stderr*, и программа завершится с кодом статуса 1.

Обсуждение

Это маленький рецепт, но он решает часто возникающую при написании

скриптов задачу. Чтобы завершить программу, часто пишут что-то такое:

```
import sys
sys.stderr.write('It failed!\n')
raise SystemExit(1)
```

Но дополнительные шаги импортирования или записи в `sys.stderr` не нужны, если вы просто предоставите сообщение в `SystemExit()`.

13.3. Парсинг аргументов командной строки

Задача

Вы хотите написать программу, которая парсит аргументы, передаваемые в командной строке (найденные в `sys.argv`).

Решение

Модуль `argparse` может быть использован для парсинга аргументов командной строки. Вот простой пример, который показывает его основные возможности:

```
# search.py
...
Гипотетический инструмент командной строки для поиска в коллекции
файлов по одному или более текстовых шаблонов.
...
import argparse
parser = argparse.ArgumentParser(description='Search some files')

parser.add_argument(dest='filenames', metavar='filename', nargs='*')

parser.add_argument('-p', '--pat', metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')

parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')

parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')
```

```
parser.add_argument('--speed', dest='speed', action='store',
                    choices=['slow', 'fast'], default='slow',
                    help='search speed')

args = parser.parse_args()

# Выводим собранные аргументы
print(args.filenames)
print(args.patterns)
print(args.verbose)
print(args.outfile)
print(args.speed)
```

Эта программа определяет парсер командной строки, который можно использовать так:

```
bash % python3 search.py -h
usage: search.py [-h] [-p pattern] [-v] [-o OUTFILE]
                  [--speed {slow,fast}] [filename [filename ...]]

Search some files

positional arguments:
  filename

optional arguments:
  -h, --help            show this help message and exit
  -p pattern, --pat pattern  text pattern to search for
  -v                   verbose mode
  -o OUTFILE           output file
  --speed {slow,fast}   search speed
```

Приведённый ниже сеанс демонстрирует, как данные показываются в программе. Понаблюдайте за выводом инструкций *print()*.

```
bash % python3 search.py foo.txt bar.txt
usage: search.py [-h] -p pattern [-v] [-o OUTFILE] [--speed {fast,slow}]
                  [filename [filename ...]]
search.py: error: the following arguments are required: -p/--pat

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt
filenames = ['foo.txt', 'bar.txt']
patterns = ['spam', 'eggs']
verbose = True
outfile = None
speed = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o result
```

```
filenames = ['foo.txt', 'bar.txt']
patterns = ['spam', 'eggs']
verbose = True
outfile = results
speed = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o result
--speed=fast
filenames = ['foo.txt', 'bar.txt']
patterns = ['spam', 'eggs']
verbose = True
outfile = results
speed = fast
```

Последующая обработка аргументов зависит от программы. Просто замените функции *print()* на что-то более интересное.

Обсуждение

Модуль *argparse* — один из самых крупных в стандартной библиотеке, у него огромное количество различных конфигурационных параметров. Этот рецепт показывает самое необходимое их подмножество, которое может быть использовано на старте, а потом расширено по необходимости.

Чтобы парсить аргументы, сначала вы создаёте экземпляр *ArgumentParser* и добавляете объявления для аргументов, которые вы хотите поддерживать, используя метод *add_argument()*. В каждом вызове *add_argument()* аргумент *dest* определяет имя атрибута, куда будет помещаться результат парсинга. Аргумент *metavar* используется при генерации сообщений помощи. Аргумент *action* определяет обработку, связанную с аргументом. Часто это будет *store* — для сохранения значения, или *append* — для сбора многих значений аргументов в список.

Следующий аргумент собирает все дополнительные аргументы командной строки в список. В примере это используется для составления списка имён файлов:

```
parser.add_argument(dest='filenames', metavar='filename', nargs='*')
```

Следующий аргумент устанавливает булевый флаг в зависимости от того, был ли предоставлен аргумент:

```
parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')
```

Следующий аргумент принимает единственное значение и сохраняет его в форме строки:

```
parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')
```

Следующее определение аргумента позволяет аргументу повторяться множество раз, и при этом все значения добавляются в список. Флаг *required* означает, что аргумент должен быть предоставлен хотя бы один раз. Использование *-r* и *--pat* означает, что оба имени аргумента приемлемы.

```
parser.add_argument('-p', '--pat', metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')
```

Наконец, следующее определение аргумента принимает значение и проверяет его на соответствие множеству возможных вариантов.

```
parser.add_argument('--speed', dest='speed', action='store',
                    choices={'slow', 'fast'}, default='slow',
                    help='search speed')
```

Когда опции переданы, вы просто выполняете метод *parser.parse()*. Он обработает значение *sys.argv* и вернёт экземпляр с результатами. Результаты для каждого аргумента помещаются в атрибут с именем, заданным в параметре *dest*, переданном в *add_argument()*.

Есть несколько подходов к парсингу аргументов командной строки. Например, вы можете вручную обработать *sys.argv* или использовать модуль *getopt* (который назван в честь библиотеки на языке С с похожим названием). Однако если вы пойдёте по этому пути, то закончите переписыванием кода, который уже предоставляет *argparse*. Вы также можете столкнуться с кодом, который использует библиотеку *optparse* для разбора аргументов. Хотя она очень похожа на *argparse*, последняя более современна, и в новых проектах стоит использовать именно её.

13.4. Запрос пароля во время выполнения

Задача

Вы пишете скрипт, который требует ввода пароля. Поскольку этот скрипт предназначен для интерактивного использования, вы бы хотели, чтобы он предлагал пользователю ввести пароль, а не жёстко прописывал пароль внутри себя.

Решение

Модуль `getpass` — то, что вам нужно. Он позволит вам без труда вывести предложение ввести пароль, не отображая вводимые символы в терминале пользователя. Вот как это делается:

```
import getpass

user = getpass.getuser()
passwd = getpass.getpass()

if svc_login(user, passwd):      # Вы должны написать svc_login()
    print('Yay!')
else:
    print('Boo!')
```

В этом примере функция `svc_login()` — это код, который вы должны написать для обработки введённого пароля. Очевидно, конкретная обработка зависит от приложения.

Обсуждение

Обратите внимание, что в показанном выше коде `getpass.getuser()` не предлагает пользователю ввести юзернейм. Вместо этого она использует текущий логин пользователя в соответствии с пользовательским окружением командной оболочки, либо, в качестве последнего варианта, в соответствии с локальной системной базой данных паролей (на платформах, поддерживающих модуль `pwd`).

Если вы хотите явно предлагать пользователю ввести юзернейм, что будет более надёжным способом, используйте встроенную функцию `input`:

```
user = input('Enter your username: ')
```

Также важно помнить, что некоторые системы могут не поддерживать скрытие вводимого пароля, которое обеспечивает метод `getpass()`. В этом случае Python сделает всё, что в его силах, чтобы предупредить вас о наличии проблем (т.е., он сообщит, что пароли будут показаны) перед продолжением выполнения.

13.5. Получение размера окна терминала

Задача

Вы хотите получить размер окна терминала, чтобы правильно отформатировать вывод вашей программы.

Решение

Используйте функцию `os.get_terminal_size()`:

```
>>> import os
>>> sz = os.get_terminal_size()
>>> sz
os终端大小(columns=80, lines=24)
>>> sz.columns
80
>>> sz.lines
24
>>>
```

Обсуждение

Есть много других возможных подходов для получения размера терминала: от чтения переменных окружения до выполнения низкоуровневых системных вызовов с использованием `ioctl()` и ТТУ. Но зачем вам всё это, если достаточно одного простого вызова функции?

13.6. Выполнение внешней команды и получение её вывода

Задача

Вы хотите выполнить внешнюю команду и собрать её вывод в строку Python.

Решение

Используйте функцию `subprocess.check_output()`. Например:

```
import subprocess
out_bytes = subprocess.check_output(['netstat', '-a'])
```

Это запустит определённую команду и вернёт ее вывод в форме байтовой строки. Если вам нужно интерпретировать получившиеся байты как текст, добавьте дополнительный шаг декодирования. Например:

```
out_text = out_bytes.decode('utf-8')
```

Если выполнение команды возвращает ненулевой код выхода, возбуждается исключение. Вот пример отлова ошибок и получения созданного вывода вместе с кодом выхода:

```
try:
    out_bytes = subprocess.check_output(['cmd', 'arg1', 'arg2'])
except subprocess.CalledProcessError as e:
    out_bytes = e.output          # Вывод, сгенерированный до ошибки
    code      = e.returncode      # Код возврата
```

По умолчанию `check_output()` возвращает только вывод, записанный в стандартный поток вывода. Если вы хотите собрать и стандартный поток вывода, и стандартный поток ошибок, используйте аргумент `stderr`:

```
out_bytes = subprocess.check_output(['cmd', 'arg1', 'arg2'],
                                   stderr=subprocess.STDOUT)
```

Если вам нужно выполнить команду с таймаутом, используйте аргумент `timeout`:

```
try:
    out_bytes = subprocess.check_output(['cmd', 'arg1', 'arg2'], timeout=5)
except subprocess.TimeoutExpired as e:
    ...

```

В обычном случае команды выполняются без помощи командной оболочки

(т.е., *sh*, *bash* и т.п.) Вместо этого предоставленный список строк передается низкоуровневой команде, такой как `os.execve()`. Если вы хотите, чтобы команда выполнялась оболочкой, передайте её в форме простой строки и добавьте аргумент `shell=True`. Это иногда полезно, если пытаетесь заставить Python выполнить сложную команду оболочки, включающую каналы, перенаправление ввода-вывода и другие возможности. Например:

```
out_bytes = subprocess.check_output('grep python | wc > out', shell=True)
```

Обратите внимание, что выполнение команд под управлением оболочки может вызвать проблемы с безопасностью, если аргументы вводятся пользователем. Функция `shlex.quote()` может быть использована в этом случае для правильной передачи аргументов для включения в команды оболочки.

Обсуждение

Функция `check_output()` — самый простой путь выполнения внешней команды и получения её вывода. Однако если вам нужно обеспечить более сложную коммуникацию с подпроцессом, такую как отсылку ему ввода, вам нужно следовать иному пути. Для этого нужно использовать класс `subprocess.Popen` напрямую. Например:

```
import subprocess

# Какой-то текст для отсылки
text = b'''
hello world
this is a test
goodbye
'''

# Запускаем команду с каналами
p = subprocess.Popen(['wc'],
                    stdout = subprocess.PIPE,
                    stdin = subprocess.PIPE)

# Посыпаем данные и получаем вывод
stdout, stderr = p.communicate(text)

# Декодируем, чтобы интерпретировать текст
out = stdout.decode('utf-8')
err = stderr.decode('utf-8')
```

Модуль `subprocess` не подходит для коммуникации с внешними командами, которые ожидают общения с настоящим TTY. Например, вы не можете использовать его для автоматизации задач, которые запрашивают у пользователя пароль (например, сессий ssh). Для этого вам нужен сторонний модуль, такой как основанный на популярном семействе инструментов “`expect`” (например, `reexpect` или похожий).

13.7. Копирование или перемещение файлов и каталогов

Задача

Вам нужно скопировать или переместить файлы и каталоги, но вы не хотите делать это через вызов команд оболочки.

Решение

Модуль `shutil` предлагает переносимую реализацию функций для копирования файлов и каталогов. Использование абсолютно прямолинейно:

```
import shutil

# Kopiruem src в dst. (cp src dst)
shutil.copy(src, dst)

# Kopiruem данные, сохраняя метаданные (cp -p src dst)
shutil.copy2(src, dst)

# Kopiruem дерево каталогов (cp -R src dst)
shutil.copytree(src, dst)

# Перемещаем src в dst (mv src dst)
shutil.move(src, dst)
```

Аргументы этих функций — это строки с именами файлов или каталогов. Лежащая в основе семантика пытается эмулировать похожие команды Unix, как показано выше в комментариях к коду.

По умолчанию эти команды переходят по символическим ссылкам. Например, если исходный файл — это символическая ссылка, тогда файл назначения будет копией файла, на который указывает ссылка. Если вы

вместо этого хотите скопировать саму символическую ссылку, предоставьте именованный аргумент *follow_symlinks* таким образом:

```
shutil.copy2(src, dst, follow_symlinks=False)
```

Если вы хотите сохранить символические ссылки в копируемых каталогах, сделайте так:

```
shutil.copytree(src, dst, symlinks=True)
```

copytree() опционально позволяет вам проигнорировать некоторые файлы и каталоги во время процесса копирования. Чтобы сделать это, вы предоставляете функцию в аргументе *ignore*, которая принимает имена каталогов и файлов в качестве ввода, и возвращает список имён, которые нужно проигнорировать, в качестве результата. Например:

```
def ignore_pyc_files(dirname, filenames):
    return [name for name in filenames if name.endswith('.pyc')]

shutil.copytree(src, dst, ignore=ignore_pyc_files)
```

Поскольку пропуск имён файлов по шаблонам — это типичная операция, есть полезная функция *ignore_patterns()*, которая берёт это на себя. Например:

```
shutil.copytree(src, dst, ignore=shutil.ignore_patterns('*~', '*.pyc'))
```

Обсуждение

Использование *shutils* для копирования файлов и каталогов по большей части прямолинейно. Однако есть один тонкий момент, касающийся метаданных файлов: функции типа *copy2()* пытаются их сохранить. Базовая информация, такая как время последнего доступа и создания, а также разрешения, сохраняется всегда, однако владельцы, ACL (списки контроля доступа), форки ресурсов и прочие расширенные метаданные могут сохраняться или не сохраняться — всё зависит от операционной системы и собственных разрешений пользователя на доступ. Вам, вероятно, не стоит использовать функцию типа *shutil.copytree()* для выполнения резервного копирования системы.

При работе с именами файлов убедитесь, что вы используете функции из

`os.path` для обеспечения высокой переносимости (особенно если программа должна работать и на Unix, и на Windows). Например:

```
>>> filename = '/Users/guido/programs/spam.py'
>>> import os.path
>>> os.path.basename(filename)
'spam.py'
>>> os.path.dirname(filename)
'/Users/guido/programs'
>>> os.path.split(filename)
('/Users/guido/programs', 'spam.py')
>>> os.path.join('/new/dir', os.path.basename(filename))
'/new/dir/spam.py'
>>> os.path.expanduser('~/guido/programs/spam.py')
'/Users/guido/programs/spam.py'
>>>
```

Есть один тонкий момент при копировании каталогов с помощью функции `copytree()` — это обработка ошибок. Например, в процессе копирования функция может встретить битые символические ссылки, файлы, к которым не будет доступа из-за проблем с разрешениями, и так далее. Чтобы разобраться с этим, все возникшие исключения собираются в список и группируются в одно исключение, которое возбуждается в конце выполнения операции. Вот как вы можете его обработать:

```
try:
    shutil.copytree(src, dst)
except shutil.Error as e:
    for src, dst, msg in e.args[0]:
        # src – имя источника
        # dst – имя назначения
        # msg – сообщение об ошибке из исключения
        print(dst, src, msg)
```

Если вы предоставите именованный аргумент `ignore_dangling_symlinks=True`, то `copytree` будет игнорировать битые символические ссылки.

В этом рецепте мы показали наиболее часто используемые функции. Однако в `shutil` очень много функций, связанных с копированием данных. На его [документацию](#) определённо стоит взглянуть.

13.8. Создание и распаковка архивов

Задача

Вам нужно создавать или распаковывать архивы распространённых форматов (например, .tar, .tgz или .zip).

Решение

В модуле *shutil* есть функции *make_archive()* и *unpack_archive()*, которые делают именно то, что вам нужно. Например:

```
>>> import shutil  
>>> shutil.unpack_archive('Python-3.3.0.tgz')  
>>> shutil.make_archive('py33', 'zip', 'Python-3.3.0')  
'~/Users/beazley/Downloads/py33.zip'  
>>>
```

Второй аргумент *make_archive()* — это нужный формат вывода. Чтобы получить список поддерживаемых форматов, используйте *get_archive_formats()*. Например:

```
>>> shutil.get_archive_formats()  
[('bz2tar', "bz2'ed tar-file"), ('gztar', "gzip'ed tar-file"),  
 ('tar', 'uncompressed tar file'), ('zip', 'ZIP file')]  
>>>
```

Обсуждение

В Python есть другие библиотечные модули для низкоуровневых операций с архивами различных форматов (например, *tarfile*, *zipfile*, *gzip*, *bz2* и т.д.) Однако если вам просто нужно создать или распаковать архив, углубляться в детали не стоит. Вы можете спокойно использовать высокоуровневые функции из *shutil*.

У этих функций есть множество различных опций для логирования, тестовых прогонов, работы с разрешениями файлов и так далее. Обратитесь к документации *shutil* за подробностями.

13.9. Поиск файлов по имени

Задача

Вам нужно написать скрипт, в котором используется поиск файлов — например, скрипт для переименования файлов или архивации логов, но вы бы предпочли не вызывать утилиты оболочки из Python. Или же вы хотите предоставить нестандартное поведение, которое не так легко обеспечить путём передачи команд в оболочку.

Решение

Чтобы искать файлы, используйте функцию `os.walk()`, предоставляя ей каталог высшего уровня. Вот пример функции, которая находит конкретное имя файла и выводит полный путь всех для всех совпадений:

```
#!/usr/bin/env python3.3
import os

def findfile(start, name):
    for relpath, dirs, files in os.walk(start):
        if name in files:
            full_path = os.path.join(start, relpath, name)
            print(os.path.normpath(os.path.abspath(full_path)))

if __name__ == '__main__':
    findfile(sys.argv[1], sys.argv[2])
```

Сохраните этот скрипт под именем `findfile.py` и запустите его из командной строки, предоставив в форме позиционных аргументов стартовую точку и имя:

```
bash % ./findfile.py . myfile.txt
```

Обсуждение

Метод `os.walk()` обходит иерархию каталогов, и для каждого каталога, в который он входит, возвращает кортеж из трёх элементов, который содержит относительный путь к инспектируемому каталогу, список со всеми именами каталогов в этом каталоге, а также список имён файлов в каталоге.

Для каждого кортежа вы просто проверяете, есть ли целевое имя файла в списке `files`. Если оно там есть, используется `os.path.join()` для сборки пути воедино. Чтобы избежать формирования странно выглядящих путей типа `././foo//bar`, для исправления результата используются две дополнительные

функции. Первая — это `os.path.abspath()`, которая принимает путь, который может быть относительным, и формирует абсолютный путь. Вторая — `os.path.normpath()`, которая нормализует путь путём исправления проблем с двойными слэшами, несколькими ссылками на текущий каталог и т.д.

Хотя этот скрипт очень прост по сравнению с возможностями утилиты `find`, которая есть на UNIX-платформах, но у него есть преимущество: кроссплатформенность. В дальнейшем в него можно без труда добавить множество функций с сохранением переносимости. В качестве примера приведём функцию, которая выводит все недавно изменённые файлы:

```
#!/usr/bin/env python3.3

import os
import time

def modified_within(top, seconds):
    now = time.time()
    for path, dirs, files in os.walk(top):
        for name in files:
            fullpath = os.path.join(path, name)
            if os.path.exists(fullpath):
                mtime = os.path.getmtime(fullpath)
                if mtime > (now - seconds):
                    print(fullpath)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: {} dir seconds'.format(sys.argv[0]))
        raise SystemExit(1)

    modified_within(sys.argv[1], float(sys.argv[2]))
```

Построение намного более сложной логики на базе этой маленькой функции не займёт много времени, если использовать различные возможности `os`, `os.path`, `glob` и других похожих модулей. См. [рецепт 5.11.](#) и [рецепт 5.13.](#)

13.10. Чтение конфигурационных файлов

Задача

Вы хотите читать конфигурационные файлы, написанные в

распространённом формате *.ini*.

Решение

Модуль *configparser* может быть использован для чтения конфигурационных файлов. Предположим, например, что у вас есть такой конфигурационный файл:

```
; config.ini
; Sample configuration file
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local

# Установки, связанные с отладочной конфигурацией
[debug]
log_errors=true
show_warnings=False

[server]
port: 8080
nworkers: 32
pid-file=/tmp/spam.pid
root=/www/root
signature:
=====
Brought to you by the Python Cookbook
=====
```

Вот пример того, как читать его и извлекать значения:

```
>>> from configparser import ConfigParser
>>> cfg = ConfigParser()
>>> cfg.read('config.ini')
['config.ini']
>>> cfg.sections()
['installation', 'debug', 'server']
>>> cfg.get('installation', 'library')
'/usr/local/lib'
>>> cfg.getboolean('debug', 'log_errors')
True
>>> cfg.getint('server', 'port')
8080
>>> cfg.getint('server', 'nworkers')
32
```

```
>>> print(cfg.get('server', 'signature'))
=====
Brought to you by the Python Cookbook
=====
>>>
```

Если захотите, вы также можете изменить конфигурацию и записать её обратно в файл с помощью метода `cfg.write()`. Например:

```
>>> cfg.set('server', 'port', '9000')
>>> cfg.set('debug', 'log_errors', 'False')
>>> import sys
>>> cfg.write(sys.stdout)
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

[debug]
log_errors = False
show_warnings = False

[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root
signature =
=====
Brought to you by the Python Cookbook
=====

>>>
```

Обсуждение

Конфигурационные файлы хорошо подходят для определения конфигурации в легкочитаемом людьми формате. Внутри каждого конфигурационного файла значения сгруппированы в разные секции (в примере это “`installation`”, “`debug`” и “`server`”). Каждая секция затем определяет значения для различных переменных, относящихся к этой секции.

Есть несколько стоящих упоминания отличий между конфигурационным файлом и использованием файла с исходным кодом Python для той же цели. Во-первых, синтаксис намного более «необязательный» и «расхлябанный». Например, эти присваивания равнозначны:

```
prefix=/usr/local
prefix: /usr/local
```

Предполагается, что имена, используемые в конфигурационном файле, также являются нечувствительными к регистру. Например:

```
>>> cfg.get('installation', 'PREFIX')
'/usr/local'
>>> cfg.get('installation', 'prefix')
'/usr/local'
>>>
```

При парсинге значений такие методы, как `getboolean()`, ищут любое разумное значение. Например, такие записи эквивалентны:

```
log_errors = true
log_errors = True
log_errors = Yes
log_errors = 1
```

Вероятно, наиболее значительное отличие между конфигурационным файлом и кодом Python в том, что, в отличие от скриптов, конфигурационные файлы не выполняются последовательно сверху вниз. Вместо этого файл читается полностью. Если делаются подстановки переменных, они будут выполнены. Например, в этой части конфигурационного файла неважно, что переменной `prefix` присваивается значение после других переменных, которые её используют:

```
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local
```

Легко проглядеть возможность `ConfigParser`, которая заключается в том, что он может читать несколько конфигурационных файлов и объединять их результаты в единую конфигурацию. Предположим, например, что пользователь создал свой собственный конфигурационный файл, который выглядит так:

```
; ~/.config.ini
[installation]
prefix=/Users/beazley/test
```

```
[debug]
log_errors=False
```

Этот файл может быть объединён с предыдущей конфигурацией путём чтения их по отдельности. Например:

```
>>> # Ранее прочитанная конфигурация
>>> cfg.get('installation', 'prefix')
'/usr/local'

>>> # Слияние в определённую пользователем конфигурацию
>>> import os
>>> cfg.read(os.path.expanduser('~/.config.ini'))
['/Users/beazley/.config.ini']
>>> cfg.get('installation', 'prefix')
'/Users/beazley/test'
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.getboolean('debug', 'log_errors')
False
>>>
```

Понаблюдайте, как переназначение переменной *prefix* влияет на другие связанные переменные, такие как установки *library*. Это работает, потому что интерполяция переменных производится настолько поздно, насколько это возможно. Вы можете увидеть это в ходе такого эксперимента:

```
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.set('installation', 'prefix', '/tmp/dir')
>>> cfg.get('installation', 'library')
'/tmp/dir/lib'
>>>
```

Наконец, важно отметить, что Python не поддерживает весь набор возможностей, которые можно обнаружить в .ini-файлах, используемых другими программами (например, Windows-приложениями). Посмотрите документацию *configparser*, чтобы ознакомиться с деталями синтаксиса и поддерживаемыми возможностями.

13.11. Добавление логирования в простые скрипты

Задача

Вы хотите, чтобы скрипты и простые программы записывали диагностическую информацию в логи.

Решение

Самый простой способ добавить логирование в простую программу — использовать модуль *logging*. Например:

```
import logging

def main():
    # Конфигурируем логирующую систему
    logging.basicConfig(
        filename='app.log',
        level=logging.ERROR
    )

    # Переменные (чтобы заставить работать следующие вызовы)
    hostname = 'www.python.org'
    item = 'spam'
    filename = 'data.csv'
    mode = 'r'

    # Примеры логирующих вызовов (вставьте в вашу программу)
    logging.critical('Host %s unknown', hostname)
    logging.error("Couldn't find %r", item)
    logging.warning('Feature is deprecated')
    logging.info('Opening file %r, mode=%r', filename, mode)
    logging.debug('Got here')

if __name__ == '__main__':
    main()
```

Пять логирующих вызовов (*critical()*, *error()*, *warning()*, *info()*, *debug()*) представляют несколько уровней серьёзности в уменьшающемся порядке. Передаваемый в *basicConfig()* аргумент *level* — это фильтр. Все сообщения, которые уходят на более низком уровне, чем указанный, будут проигнорированы.

Аргумент каждой логирующей операции — это строка сообщения, за которой следуют ноль или более аргументов. При создании финального сообщения для отправки в лог, используется оператор *%* для форматирования строки сообщения с применением предоставленных аргументов.

Если вы запустите эту программу, содержание файла `app.log` будет таким:

```
CRITICAL:root:Host www.python.org unknown
ERROR:root:Could not find 'spam'
```

Если вы хотите изменить вывод или уровень вывода, вы можете поменять передаваемые в вызов `basicConfig()` параметры. Например:

```
logging.basicConfig(
    filename='app.log',
    level=logging.WARNING,
    format='%(levelname)s:%(asctime)s:%(message)s')
```

В результате вывод изменится на такой:

```
CRITICAL:2012-11-20 12:27:13,595:Host www.python.org unknown
ERROR:2012-11-20 12:27:13,595:Could not find 'spam'
WARNING:2012-11-20 12:27:13,595:Feature is deprecated
```

Как показано выше, конфигурация логирования жестко закодирована в программе. Если вы хотите конфигурировать её конфигурационным файлом, измените вызов `basicConfig()` на следующий:

```
import logging
import logging.config

def main():
    # Конфигурируем логирующую систему
    logging.config.fileConfig('logconfig.ini')
    ...
```

Теперь создайте конфигурационный файл `logconfig.ini`, который выглядит так:

```
[loggers]
keys=root

[handlers]
keys=defaultHandler

[formatters]
keys=defaultFormatter

[logger_root]
level=INFO
handlers=defaultHandler
```

```
qualname=root

[handler_defaultHandler]
class=FileHandler
formatter=defaultFormatter
args=('app.log', 'a')

[formatter_defaultFormatter]
format=%(levelname)s:%(name)s:%(message)s
```

Если вы хотите внести изменения в конфигурацию, просто отредактируйте *logconfig.ini*.

Обсуждение

Если мы на секунду забудем, что модуль *logging* понимает миллион продвинутых конфигурационных параметров, то это решение выглядит вполне достаточным для простых программ и скриптов. Просто убедитесь, что вы выполняете вызов *basicConfig()* перед тем, как производить любые логирующие вызовы — и ваша программа будет генерировать логирующий вывод.

Если вы хотите перенаправить сообщения логирования в стандартный поток ошибок, а не в файл, то не передавайте никакой информации об имени файла в *basicConfig()*. Например, просто сделайте так:

```
logging.basicConfig(level=logging.INFO)
```

Тонкость использования *basicConfig()* в том, что её можно вызвать в программе только один раз. Если вам позже нужно будет изменить конфигурацию модуля *logging*, вам нужно получить корневой логгер и изменить его напрямую. Например:

```
logging.getLogger().level = logging.DEBUG
```

Нужно подчеркнуть, что этот рецепт демонстрирует только базовое использование модуля *logging*. Его можно настроить под конкретные потребности намного более продвинутым образом. Рекомендуем обратиться к отличному ресурсу [Logging Cookbook](#).

13.12. Добавление логирования в

библиотеки

Задача

Вы хотите добавить возможность логирования в библиотеку, но не хотите, чтобы оно вмешивалось в программы, которые не используют логирование.

Решение

Для библиотек, которым нужно осуществлять логирование, вы можете создать выделенный объект логгера и инициализировать его таким образом:

```
# somelib.py

import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

# Пример функции (для тестирования)
def func():
    log.critical('A Critical Error!')
    log.debug('A debug message')
```

С такой конфигурацией по умолчанию логирование вестись не будет.

Например:

```
>>> import somelib
>>> somelib.func()
>>>
```

Но если система логирования будет сконфигурирована, начнут появляться сообщения лога. Например:

```
>>> import logging
>>> logging.basicConfig()
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>>
```

Обсуждение

Библиотеки представляют собой особую проблему для логирования,

поскольку неизвестна информация об окружении, в котором они используются. В качестве общего правила: никогда не пишите библиотечный код таким образом, чтобы он пытался сконфигурировать систему логирования самостоятельно, или же пытался делать предположения об уже существующей конфигурации логирования. Вам нужно соблюдать осторожность и обеспечить изоляцию.

Вызов `getLogger(__name__)` создает объект-логгер, который имеет то же имя, что и вызывающий модуль. Поскольку все модули уникальны, это создаст выделенный логгер, который, скорее всего, будет отделен от других логгеров.

Операция `log.addHandler(logging.NullHandler())` прикрепляет пустой (null) обработчик к только что созданному объекту-логгеру. Пустой обработчик по умолчанию игнорирует все сообщения логирования. Так что если библиотека используется, а логирование не было сконфигурировано, то сообщения или предупреждения появляться не будут.

Тонкий момент этого рецепта заключается в том, что логирование отдельных библиотек может быть сконфигурировано независимо, несмотря на другие установки логирования. Например, рассмотрим такой код:

```
>>> import logging
>>> logging.basicConfig(level=logging.ERROR)
>>> import somelib
>>> somelib.func()
CRITICAL:somelib:A Critical Error!

>>> # Изменяем уровень логирования только для 'somelib'
>>> logging.getLogger('somelib').level=logging.DEBUG
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
DEBUG:somelib:A debug message
>>>
```

Здесь корневой логгер был сконфигурирован таким образом, чтобы выводить только сообщения уровня ERROR или выше. Однако уровень логгера библиотеки `somelib` был отдельно сконфигурирован так, чтобы выводить отладочные сообщения. Эта установка преодолевает глобальную.

Способность менять установки логирования отдельного модуля может быть полезна в качестве инструмента отладки, поскольку вам не нужно менять глобальные установки логирования — просто измените уровень для одного

модуля, от которого вы хотите видеть больше сообщений.

В [Logging HOWTO](#) вы найдёте много сведений о модуле *logging* и другие полезные советы.

13.13. Создание таймера-секундомера

Задача

Вы хотите записывать время выполнения различных задач.

Решение

В модуле *time* есть много различных функций для работы со временем. Однако часто бывает полезно натянуть на них высокоуровневый интерфейс, который имитирует функциональность секундомера. Например:

```
import time

class Timer:
    def __init__(self, func=time.perf_counter):
        self.elapsed = 0.0
        self._func = func
        self._start = None

    def start(self):
        if self._start is not None:
            raise RuntimeError('Already started')
        self._start = self._func()

    def stop(self):
        if self._start is None:
            raise RuntimeError('Not started')
        end = self._func()
        self.elapsed += end - self._start
        self._start = None

    def reset(self):
        self.elapsed = 0.0

    @property
    def running(self):
        return self._start is not None

    def __enter__(self):
```

```
    self.start()
    return self

def __exit__(self, *args):
    self.stop()
```

Этот класс определяет таймер, который может быть запущен, остановлен и сброшен по указанию пользователя. Он следит за прошедшим временем, сохраняя его в атрибуте `elapsed`. Вот пример того, как его можно использовать:

```
def countdown(n):
    while n > 0:
        n -= 1

# Пример 1: Явный запуск/остановка
t = Timer()
t.start()
countdown(1000000)
t.stop()
print(t.elapsed)

# Пример 2: Как менеджер контекста
with t:
    countdown(1000000)
print(t.elapsed)

with Timer() as t2:
    countdown(1000000)
print(t2.elapsed)
```

Обсуждение

Этот рецепт предоставляет простой, но весьма полезный класс для измерения таймингов и отслеживания прошедшего времени. Он также является неплохим примером того, как реализовать поддержку протокола менеджера контекста и инструкции `with`.

Проблема с выполнением тайминга касается лежащей в основе функции. В качестве общего правила: точность измерений времени, выполненного с помощью функций типа `time.time()` или `time.clock()`, варьируется в зависимости от операционной системы. И наоборот: функция `time.perf_counter()` всегда использует доступный в системе таймер, имеющий максимальную точность.

Как показано выше, с помощью класса `Timer` время записывается в

соответствии со «внешним» временем и включает всё то время, которое программа спала. Если вы хотите учитывать только время, когда процесс использовал CPU, используйте `time.process_time()`. Например:

```
t = Timer(time.process_time)
with t:
    countdown(1000000)
print(t.elapsed)
```

Функции `time.perf_counter()` и `time.process_time()` возвращают «время» в долях секунды. Однако конкретное значение времени не имеет какого-либо особенного смысла. Чтобы придать результатам смысл, вы должны вызывать функции дважды и вычислить разницу.

Дополнительные примеры тайминга и профилирования вы найдёте в [рецепте 14.13](#).

13.14. Установка лимитов на использование памяти и CPU

Задача

Вы хотите установить некие ограничения на использование памяти и процессора программой, выполняющейся в Unix.

Решение

Модуль `resource` может быть использован для выполнения обеих этих задач. Например, чтобы ограничить потребление CPU, сделайте так:

```
import signal
import resource
import os

def time_exceeded(signo, frame):
    print("Time's up!")
    raise SystemExit(1)

def set_max_runtime(seconds):
    # Устанавливает обработчик сигнала и лимит ресурса
    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard))
```

```
signal.signal(signal.SIGXCPU, time_exceeded)

if __name__ == '__main__':
    set_max_runtime(15)
    while True:
        pass
```

При запуске этого кода по истечении времени будет сгенерирован сигнал *SIGXCPU*. Затем программа очистится и будет произведен выход.

Чтобы ограничить потребление памяти, установите лимит на размер используемого адресного пространства в целом. Например:

```
import resource

def limit_memory(maxsize):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (maxsize, hard))
```

Когда лимит установлен, программы начнут генерировать исключения *MemoryError*, если доступная память кончится.

Обсуждение

В этом рецепте функция *setrlimit()* используется для установки мягкого и жёсткого лимитов на конкретный ресурс. Мягкий лимит — это значение, по достижению которого операционная система в типичном случае ограничит процесс или уведомит его путём отправки сигнала. Жёсткий лимит определяет верхнюю границу значений, которые могут быть использованы для мягкого лимита. Обычно это контролируется параметром системного уровня, установленного системным администратором. Хотя жёсткий лимит может быть понижен, он не может быть повышен процессами пользователя (даже если процесс сам его понизил).

Функция *setrlimit()* может быть дополнительно использована для установки лимитов на такие вещи, как количество процессов-потомков, количество открытых файлов и прочие подобные системные ресурсы. Обратитесь к документации модуля *resource* за дополнительными сведениями.

Обратите внимание, что этот рецепт работает только на Unix-системах, причём не на всех. Например, когда это тестировалось, то примеры работали на Linux, но не на OS X.

13.15. Запуск браузера

Задача

Вы хотите запустить браузер из скрипта и заставить его открыть указанный вами URL.

Решение

Для платформонезависимого запуска браузера можно использовать модуль `webbrowser`. Например:

```
>>> import webbrowser  
>>> webbrowser.open('http://www.python.org')  
True  
>>>
```

Это откроет запрошенную страницу с помощью дефолтного браузера. Если вы хотите получить больше контроля над открытием страницы, вы можете использовать одну из следующих функций:

```
>>> # Открыть страницу в новом окне браузера  
>>> webbrowser.open_new('http://www.python.org')  
True  
>>>  
  
>>> # Открыть страницу в новой вкладке браузера  
>>> webbrowser.open_new_tab('http://www.python.org')  
True  
>>>
```

Эти функции пытаются открыть страницу в новом окне браузера или в табе, если это возможно и поддерживается браузером.

Если вы хотите открыть страницу в конкретном браузере, вы можете использовать функцию `webbrowser.get()`, чтобы определить конкретный браузер. Например:

```
>>> c = webbrowser.get('firefox')  
>>> c.open('http://www.python.org')  
True  
>>> c.open_new_tab('http://docs.python.org')  
True
```

>>>

Полный список названий поддерживаемых браузеров вы можете найти в [документации Python](#).

Обсуждение

Возможность легко запустить браузер может оказаться весьма кстати в некоторых скриптах. Например, скрипт может выполнять некий деплоймент на сервер, и вы хотите заставить его также быстро запустить браузер, чтобы удостовериться, что всё работает. Или же программа может записывать данные в форме HTML-страниц, и вы просто хотите сразу посмотреть результат. В обоих случаях модуль *webbrowser* будет простым решением.

14. Тестирование, отладка и исключения

Тестирование — это круто, но отладка?.. Не особо. Тот факт, что у нас нет компилятора, который проанализировал бы код перед тем, как Python его выполнит, делает тестирование важнейшей частью разработки. Цель этой главы — обсудить некоторые распространённые проблемы, связанные с тестированием, отладкой и обработкой исключений. Мы не будем давать введение в разработку, управляемую тестированием (TDD), или модуль *unittest*. Мы подразумеваем, что вы уже в какой-то степени знакомы с концепциями тестирования.

14.1. Тестирование отправки вывода в `stdout`

Задача

У вас есть программа, в которой есть метод, чей вывод должен идти в стандартный поток вывода (`sys.stdout`). Это практически всегда означает, что он выводит текст на экран. Вы хотите написать тест для вашего кода, который это проверяет: что при предоставлении корректного ввода показывается правильный вывод.

Решение

Используя функцию `patch()` из модуля `unittest.mock`, можно легко сделать мок (имитировать) `sys.stdout` для одного теста, а затем вернуть всё обратно — без использования запутанных временных переменных или протечек замоканного (имитированного) состояния между тест-кейсами.

Рассмотрим, например, следующую функцию в модуле `mymodule`:

```
# mymodule.py

def urlprint(protocol, host, domain):
    url = '{}://{}.{}'.format(protocol, host, domain)
    print(url)
```

Встроенная функция `print` по умолчанию посыпает вывод в `sys.stdout`. Чтобы проверить, попадает ли вывод туда на самом деле, вы можете замокать его с помощью подменяющего объекта, а затем сделать асsertы (удостоверяющие проверки) на предмет того, что происходит. Метод `patch()` из `unittest.mock` позволяет удобно заменить объекты только в контексте выполнения теста, возвращая всё к изначальному состоянию сразу после завершения теста. Вот тестирующий код для `mymodule`:

```
from io import StringIO
from unittest import TestCase
from unittest.mock import patch
import mymodule

class TestURLPrint(TestCase):
    def test_url_gets_to_stdout(self):
        protocol = 'http'
        host = 'www'
        domain = 'example.com'
        expected_url = '{}://{}.{}\n'.format(protocol, host, domain)

        with patch('sys.stdout', new=StringIO()) as fake_out:
            mymodule.urlprint(protocol, host, domain)
            self.assertEqual(fake_out.getvalue(), expected_url)
```

Обсуждение

Функция `urlprint()` принимает три аргумента, и тест начинается с создания фейковых (модельных, временных) аргументов. Переменной `expected_url`

присваивается строка, содержащая ожидаемый вывод.

Чтобы запустить тест, функция `unittest.mock.patch()` используется в качестве менеджера контекста, чтобы заменить значение `sys.stdout` объектом `StringIO`. Переменная `fake_out` — это мок-объект, который создается по ходу дела. Он может быть использован внутри тела блока инструкции `with`, чтобы выполнять различные проверки. Когда выполнение блока инструкции `with` завершается, `patch` возвращает всё к состоянию, которое имело место до запуска теста.

Стоит отметить, что некоторые расширения на С для Python могут писать напрямую в стандартный поток вывода, обходя настройку `sys.stdout`. Этот рецепт не поможет справиться с таким сценарием, но должен отлично работать с чистым кодом на Python (если вам нужно захватывать вывод от таких расширений на С, то вы можете сделать это путём открытия временного файла и выполнения различных трюков с файловыми дескрипторами, с помощью которых стандартный вывод временно перенаправляется в этот файл).

Дополнительную информацию о захвате ввода-вывода в строки и объекты `StringIO` вы можете найти в [рецепте 5.6](#).

14.2. Патчинг объектов в юнит-тестах

Задача

Вы пишете юнит-тесты, и вам нужно применить патчи к выбранным объектам, чтобы создать ассерты по поводу того, как они будут использованы в teste (например, ассерты о вызове с определёнными параметрами, доступе к определённым атрибутам и т.п.)

Решение

Для решения этой задачи можно использовать функцию `unittest.mock.patch()`. Это немного необычно, но `patch()` можно использовать и как декоратор, и отдельно. Например, вот пример её использования в качестве декоратора:

```
from unittest.mock import patch
import example
```

```
@patch('example.func')
def test1(x, mock_func):
    example.func(x)                      # Использует пропатченную example.func
    mock_func.assert_called_with(x)
```

Также она может быть использована в качестве менеджера контекста:

```
with patch('example.func') as mock_func:
    example.func(x)      # Использует пропатченную example.func
    mock_func.assert_called_with(x)
```

И, наконец, вы можете пропатчить что-то вручную:

```
p = patch('example.func')
mock_func = p.start()
example.func(x)
mock_func.assert_called_with(x)
p.stop()
```

Если нужно, вы можете последовательно применить декораторы и менеджеры контекста, чтобы пропатчить несколько объектов. Например:

```
@patch('example.func1')
@patch('example.func2')
@patch('example.func3')
def test1(mock1, mock2, mock3):
    ...

def test2():
    with patch('example.patch1') as mock1, \
        patch('example.patch2') as mock2, \
        patch('example.patch3') as mock3:
    ...
```

Обсуждение

patch() работает так: берёт существующий объект с полностью определённым именем, которое вы предоставите, и заменяет его новым значением.

Изначальное значение затем восстанавливается после завершения выполнения декорированной функции или выхода из менеджера контекста. По умолчанию значения заменяются экземплярами *MagicMock*. Например:

```
>>> x = 42
>>> with patch('__main__.x'):
```

```
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

Однако вы можете заменить значение чем угодно — достаточно лишь предоставить этот объект в качестве второго аргумента в `patch()`:

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

Экземпляры `MagicMock` обычно используются в качестве подменных значений, предназначенных для имитации вызываемых объектов и экземпляров. Они записывают информацию об использовании и позволяют вам делать асsertы. Например:

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>

>>> m.upper.return_value = 'HELLO'
>>> m.upper('hello')
'HELLO'
>>> assert m.upper.called
>>> m.split.return_value = ['hello', 'world']
>>> m.split('hello world')
['hello', 'world']
>>> m.split.assert_called_with('hello world')
>>>
```

```
>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>
```

Обычно такие операции выполняются в юнит-тестах. Предположим, например, что у вас есть такая функция:

```
# example.py
from urllib.request import urlopen
import csv

def dowprices():
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=sll')
    lines = (line.decode('utf-8') for line in u)
    rows = (row for row in csv.reader(lines) if len(row) == 2)
    prices = { name:float(price) for name, price in rows }
    return prices
```

В обычном случае эта функция использует `urlopen()` для получения данных из интернета и парсит их. Чтобы протестировать её с помощью юнит-теста, вы можете захотите предоставить ей более предсказуемый набор данных, который вы сами создадите. Вот пример, использующий патчинг:

```
import unittest
from unittest.mock import patch
import io
import example

sample_data = io.BytesIO(b'''\
"IBM",91.1\r
"AA",13.25\r
"MSFT",27.72\r
\r
''')

class Tests(unittest.TestCase):
    @patch('example.urlopen', return_value=sample_data)
    def test_dowprices(self, mock_urlopen):
        p = example.dowprices()
        self.assertTrue(mock_urlopen.called)
        self.assertEqual(p,
                        {'IBM': 91.1,
                         'AA': 13.25,
                         'MSFT' : 27.72})
```

```
if __name__ == '__main__':
    unittest.main()
```

В этом примере функция `urlopen()` в модуле `example` заменена объектом-моком, который возвращает `BytesIO()`, содержащий созданные заранее данные.

Важный и тонкий момент этого теста — это патчинг `example.urlopen` вместо `urllib.request.urlopen`. Когда вы применяете патчи, вы должны использовать имена, которые уже используются в тестируемом коде. Поскольку код примера использует `from urllib.request import urlopen`, функция `urlopen()`, использованная в функции `downprices()`, на самом деле находится в `example`.

Этот рецепт всего лишь слегка намёкает на то, что можно делать с помощью модуля `unittest.mock`. Обязательно прочитайте [официальную документацию](#), чтобы узнать о более продвинутых возможностях.

14.3. Проверка вызывающих исключения условий в рамках юнит-тестов

Задача

Вы хотите написать юнит-тест, который чётко показывает, возбуждается ли исключение.

Решение

Чтобы провести тест на исключения, используйте метод `assertRaises()`. Например, если вы хотите проверить, возбуждает ли функция `ValueError`, используйте такой код:

```
import unittest

# Простая функция для примера
def parse_int(s):
    return int(s)

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaises(ValueError, parse_int, 'N/A')
```

Если вам нужно как-то проверить значение исключения, тогда нужен другой подход. Например:

```
import errno

class TestIO(unittest.TestCase):
    def test_file_not_found(self):
        try:
            f = open('/file/not/found')
        except IOError as e:
            self.assertEqual(e.errno, errno.ENOENT)
```

Обсуждение

Метод `assertRaises()` предоставляет удобный способ провести проверку на наличие исключения. Обычная проблема — написать тесты, которые вручную пытаются работать с исключениями. Например:

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
```

Проблема такого подхода в том, что легко забыть о граничных случаях, таких как «что будет, если исключение не возбуждается?». Вам нужно добавить дополнительную проверку для такой ситуации, как показано тут:

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
        else:
            self.fail('ValueError not raised')
```

Метод `assertRaises()` просто берёт на себя заботу об этих деталях, так что лучше использовать именно его.

Единственное ограничение `assertRaises()` в том, что она не предоставляет средства для проверки значения создаваемого объекта исключения. Чтобы сделать это, вам нужно вручную проверить его, как показано выше. Иногда

между двумя этим крайними точками вы можете задуматься об использовании метода `assertRaisesRegex()`, который позволяет вам одновременно проводить проверку на исключение и выполнять поиск совпадений по регулярному выражению на строковом представлении исключения. Например:

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaisesRegex(ValueError, 'invalid literal .*',
                             parse_int, 'N/A')
```

Малоизвестный факт об `assertRaises()` и `assertRaisesRegex()` — они могут быть использованы в качестве менеджеров контекста:

```
class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        with self.assertRaisesRegex(ValueError, 'invalid literal .*'):
            r = parse_int('N/A')
```

Эта форма может быть полезна, если в вашем тесте много шагов (например, предустановка), помимо простого выполнения вызываемого объекта.

14.4. Логирование вывода теста в файл

Задача

Вы хотите, чтобы результаты прогона юнит-тестов записались в файл, а не выводились в стандартный поток вывода.

Решение

Очень распространённый приём запуска юнит-тестов — включить в конец файла с тестами вот такой небольшой фрагмент кода:

```
import unittest

class MyTest(unittest.TestCase):
    ...

if __name__ == '__main__':
    unittest.main()
```

Это делает файл исполняемым и выводит результаты выполняющихся тестов в стандартный поток вывода. Если вы хотите перенаправить этот вывод, вам нужно развернуть вызов `main()` и написать вашу собственную функцию `main()`:

```
import sys
def main(out=sys.stderr, verbosity=2):
    loader = unittest.TestLoader()
    suite = loader.loadTestsFromModule(sys.modules['__name__'])
    unittest.TextTestRunner(out, verbosity=verbosity).run(suite)

if __name__ == '__main__':
    with open('testing.out', 'w') as f:
        main(f)
```

Обсуждение

В этом рецепте не так интересно перенаправление результатов теста в файл, как тот факт, что это показывает нам некоторые стоящие внимания внутренние механизмы модуля `unittest`.

На базовом уровне модуль `unittest` работает так: сначала собирается комплект тестов (`test suite`). Этот комплект состоит из различных тестирующих методов, которые вы определили. Когда комплект тестов собран, выполняются тесты, из которых он состоит.

Эти две составляющие юнит-тестирования отделены друг от друга. Создаваемый в показанном выше решении экземпляр `unittest.TestLoader` используется для сборки комплекта тестов. `loadTestsFromModule()` — один из нескольких методов, которые он определяет для сбора тестов. В этом случае он сканирует модуль на наличие классов `TestCase` и извлекает из них тестовые методы. Если вы хотите чего-то более настраиваемого, метод `loadTestsFromTestCase()` (он нами не показан) может быть использован, чтобы вытянуть тестовые методы из отдельного класса, наследующего от `TestCase`.

Класс `TextTestRunner` — это пример класса, выполняющего тесты (тест-раннера). Главное назначение этого класса — выполнить тесты, содержащиеся в комплекте тестов. Данный класс — это тот же тест-раннер, который скрывается за функцией `unittest.main()`. Однако здесь мы передаём ей низкоуровневые параметры конфигурации, включая файл для вывода и указание включить повышенную «многословность».

Хотя этот рецепт состоит всего лишь из нескольких строк кода, он даёт

намёк на то, как вы можете далее настраивать под свои потребности фреймворк *unittest*. Чтобы кастомизировать сборку тест-комплектов, вы можете выполнить различные операции, применяя класс *TestLoader*. Чтобы кастомизировать выполнение тестов, вы можете создать собственные классы (тест-раннеры), которые эмулируют функциональность *TestRunner*. Освещение обеих тем находится за пределами возможностей этой книги, однако документация модуля *unittest* покрывает все лежащие в основе протоколы.

14.5. Пропуск или ожидание провалов тестов

Задача

Вы хотите пропустить или пометить выбранные тесты как «ожидается провал».

Решение

В модуле *unittest* есть декораторы, которые могут быть применены к выбранным тестовым методам для контролирования их обработки. Например:

```
import unittest
import os
import platform

class Tests(unittest.TestCase):
    def test_0(self):
        self.assertTrue(True)

    @unittest.skip('skipped test')
    def test_1(self):
        self.fail('should have failed')

    @unittest.skipIf(os.name=='posix', 'Not supported on Unix')
    def test_2(self):
        import winreg

    @unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific te
    def test_3(self):
        self.assertTrue(True)
```

```
@unittest.expectedFailure
def test_4(self):
    self.assertEqual(2+2, 5)

if __name__ == '__main__':
    unittest.main()
```

Если вы запустите этот код на Mac, то получите такой вывод:

```
bash % python3 testsample.py -v
test_0 (__main__.Tests) ... ok
test_1 (__main__.Tests) ... skipped 'skipped test'
test_2 (__main__.Tests) ... skipped 'Not supported on Unix'
test_3 (__main__.Tests) ... ok
test_4 (__main__.Tests) ... expected failure

-----
Ran 5 tests in 0.002s

OK (skipped=2, expected failures=1)
```

Обсуждение

Декоратор `skip()` может быть использован для пропуска теста, который вы не хотите выполнять. `skipIf()` и `skipUnless()` могут быть полезны для написания тестов, которые применяются только на некоторых платформах, версиях Python или при наличии каких-то других зависимостей. Использование декоратора `@expectedFailure` позволяет отметить тесты, о которых вы знаете, что они провалятся, и вы не хотите, чтобы тестовый фреймворк выводил информацию о них.

Декораторы для пропуска методов также могут быть применены к классам тестов целиком. Например:

```
@unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific tests'
class DarwinTests(unittest.TestCase):
    ...
```

14.6. Обработка множественных исключений

Задача

У вас есть код, который может выбросить одно из нескольких различных исключений, и вам нужно принимать во внимание все исключения, которые могут быть возбуждены, причем без дублирования кода или длинных, запутанных условий.

Решение

Вы можете обработать различные исключения с помощью единственного блока кода, объединив их в кортеж:

```
try:  
    client_obj.get_url(url)  
except (URLError, ValueError, SocketTimeout):  
    client_obj.remove_url(url)
```

В предыдущем примере метод `remove_url()` будет вызван, если возникнет любое из перечисленных исключений. Если же, однако, вам нужно обработать одно из исключений по-другому, поместите его в его собственное условие `except`:

```
try:  
    client_obj.get_url(url)  
except (URLError, ValueError):  
    client_obj.remove_url(url)  
except SocketTimeout:  
    client_obj.handle_url_timeout(url)
```

Многие исключения сгруппированы в иерархию наследования. Такие исключения вы можете поймать путём указания только базового класса. Например, вместо такого кода:

```
try:  
    f = open(filename)  
except (FileNotFoundException, PermissionError):  
    ...
```

...вы можете переписать инструкцию `except` так:

```
try:  
    f = open(filename)  
except OSError:
```

...

Это работает, поскольку `OSError` — это базовый класс, который является общим для исключений `FileNotFoundException` и `PermissionError`.

Обсуждение

Хотя это не является специфичным для обработки множественных исключений как таковых, стоит отметить, что вы можете получить «ручку» (`handler`) к выброшенному исключению, используя ключевое слово `as`:

```
try:
    f = open(filename)
except OSError as e:
    if e.errno == errno.ENOENT:
        logger.error('File not found')
    elif e.errno == errno.EACCES:
        logger.error('Permission denied')
    else:
        logger.error('Unexpected error: %d', e.errno)
```

В этом примере в переменной `e` лежит экземпляр возбуждённого исключения `OSError`. Это полезно, если вам нужно как-то инспектировать исключение — например, обработать его в зависимости от значения дополнительного кода статуса.

Помните, что условия `except` проверяются в порядке перечисления, и выполняется первое совпадение. Это может показаться немного безумным, но вы можете легко создать ситуации, в которых несколько условий `except` могут совпадать. Например:

```
>>> f = open('missing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundException: [Errno 2] No such file or directory: 'missing'
>>> try:
...     f = open('missing')
... except OSError:
...     print('It failed')
... except FileNotFoundException:
...     print('File not found')
...
It failed
>>>
```

Здесь условие `except` исключения `FileNotFoundException` не выполняется, поскольку `OSError` является более общим и поэтому совпадает с исключением `FileNotFoundException`, а перечислено в списке первым.

Совет для отладки: если вы не полностью уверены, что знаете иерархию классов конкретного исключения, то можете быстро просмотреть её путём изучения атрибута исключения `__mro__`. Например:

```
>>> FileNotFoundError.__mro__
(<class 'FileNotFoundException'>, <class 'OSError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
>>>
```

Все перечисленные классы до `BaseException` могут быть использованы с инструкцией `except`.

14.7. Ловим все исключения

Задача

Вы хотите написать код, который сможет поймать все исключения.

Решение

Чтобы поймать все исключения, напишите обработчик для `Exception`:

```
try:
    ...
except Exception as e:
    ...
    log('Reason:', e)           # Important!
```

Так вы поймаете все исключения, кроме `SystemExit`, `KeyboardInterrupt`, и `GeneratorExit`. Если вы хотите поймать и их, поменяйте `Exception` на `BaseException`.

Обсуждение

Отлов всех исключений иногда используется в качестве костыля программистами, которые не могут запомнить все возможные исключения,

которые могут возникнуть в сложных операциях. При неосторожном обращении это отличный способ написать код, который невозможно отдебажить.

При таком подходе критически важно логировать или посыпать куда-то отчёты об актуальной причине исключения (например, в файл лога, выводимые на экран сообщения об ошибках и т.п.) Если вы не сделаете этого, ваша голова взорвётся. Рассмотрим такой пример:

```
def parse_int(s):
    try:
        n = int(v)
    except Exception:
        print("Couldn't parse")
```

Если вы попробуете применить эту функцию, то получите что-то такое:

```
>>> parse_int('n/a')
Couldn't parse
>>> parse_int('42')
Couldn't parse
>>>
```

В этой точке вы можете начинать чесать голову и размышлять о том, почему оно не работает. А теперь предположим, что функция написана так:

```
def parse_int(s):
    try:
        n = int(v)
    except Exception as e:
        print("Couldn't parse")
        print('Reason:', e)
```

В этом случае вы получите вывод, который указывает на ошибку:

```
>>> parse_int('42')
Couldn't parse
Reason: global name 'v' is not defined
>>>
```

При прочих равных, вероятно, лучше быть максимально точным в вашей обработке исключений. Однако если вам нужно поймать все исключения, то убедитесь, что вы выдаёте хорошую диагностическую информацию или распространяете исключение так, что оно не теряется.

14.8. Создание собственных исключений

Задача

Вы создаёте приложение и хотите обернуть низкоуровневые исключения своими собственными, которые будут иметь больше смысла в контексте вашего приложения.

Решение

Создавать новые исключения нетрудно — просто определите их как классы, которые наследуют от *Exception* (или одного из других существующих типов исключений, если это имеет больше смысла). Например, если вы пишете код, связанный с сетевым программированием, вы могли бы определить какие-то собственные исключения:

```
class NetworkError(Exception):
    pass
class HostnameError(NetworkError):
    pass
class TimeoutError(NetworkError):
    pass
class ProtocolError(NetworkError):
    pass
```

Пользователи могут применять эти исключения обычным способом.

Например:

```
try:
    msg = s.recv()
except TimeoutError as e:
    ...
except ProtocolError as e:
    ...
```

Обсуждение

Собственные классы исключений должны практически всегда наследовать от встроенного класса *Exception*, или же наследовать от какого-то локально определённого базового исключения, которое само наследует от *Exception*. Хотя все исключения также происходят от *BaseException*, вы не должны

использовать его в качестве базового класса для новых исключений. *BaseException* зарезервирован для системных исключений выхода, таких как *KeyboardInterrupt* или *SystemExit*, и других исключений, которые должны сигнализировать приложениям о выходе. Поэтому эти исключения не предназначены для того, чтобы их отлавливать. Следование этому соглашению приводит к тому, что наследование от *BaseException* приведёт к тому, что ваши собственные кастомные исключения не будут пойманы, а это, в свою очередь, приведёт к отправке сигнала на немедленное завершение приложения!

Если у вас в коде есть кастомные исключения, и вы используете их так, как показано, то код вашего приложения будет более последовательным в глазах того, кому придётся его читать. Есть тонкий момент в проектировании, который касается группировки кастомных исключений через наследование. В сложных приложениях может иметь смысл ввести дополнительные базовые классы, которые группируют разные классы исключений. Это даёт пользователю возможность поймать конкретную ошибку:

```
try:  
    s.send(msg)  
except ProtocolError:  
    ...
```

Это также даёт возможность поймать широкий спектр ошибок:

```
try:  
    s.send(msg)  
except NetworkError:  
    ...
```

Если вы определите новое исключение, которое перегружает метод *__init__()* класса *Exception*, убедитесь, что вы всегда вызываете *Exception.__init__()* со всеми переданными аргументами. Например:

```
class CustomError(Exception):  
    def __init__(self, message, status):  
        super().__init__(message, status)  
        self.message = message  
        self.status = status
```

Это может выглядеть немного странно, но поведение *Exception* по умолчанию таково, что он принимает все переданные аргументы и сохраняет их в

атрибуте `.args` в форме кортежа. Различные библиотеки и компоненты Python ожидают, что все исключения будут иметь атрибут `.args`, так что если вы пропустите этот шаг, то, возможно, обнаружите, что ваше новое исключение не ведёт себя правильно в некоторых обстоятельствах. Чтобы продемонстрировать использование `.args`, рассмотрим интерактивный сеанс со встроенным исключением `RuntimeError`. Обратите внимание, как любое количество аргументов может быть использовано с инструкцией `raise`:

```
>>> try:  
...     raise RuntimeError('It failed')  
... except RuntimeError as e:  
...     print(e.args)  
...  
('It failed',)  
>>> try:  
...     raise RuntimeError('It failed', 42, 'spam')  
... except RuntimeError as e:  
...     print(e.args)  
...  
('It failed', 42, 'spam')  
>>>
```

За дополнительной информацией о создании собственных исключений обратитесь к [документации Python](#).

14.9. Возбуждение исключения в ответ на другое исключение

Задача

Вы хотите возбуждать исключение в ответ на поимку другого исключения, но при этом хотите добавить в информацию об обоих исключениях в отладочную информацию стека вызовов (трейсбэк, traceback).

Решение

Чтобы создать цепочку исключений, используйте инструкцию `raise from` вместо простой `raise`. Это даст вам информацию об обеих ошибках. Например:

```
>>> def example():
```

```
... try:  
...     int('N/A')  
... except ValueError as e:  
...     raise RuntimeError('A parsing error occurred') from e...  
>>>  
example()  
Traceback (most recent call last):  
  File "<stdin>", line 3, in example  
ValueError: invalid literal for int() with base 10: 'N/A'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    File "<stdin>", line 5, in example  
RuntimeError: A parsing error occurred  
>>
```

Как вы можете видеть в трейсбэке, захвачены оба исключения. Чтобы поймать такое исключение, вы могли бы использовать обычную инструкцию `except`. Однако, если пожелаете, вы можете взглянуть на атрибут `__cause__` объекта исключения, чтобы проследить цепочку исключений. Например:

```
try:  
    example()  
except RuntimeError as e:  
    print("It didn't work:", e)  
    if e.__cause__:  
        print('Cause:', e.__cause__)
```

Неявная форма цепочек исключений возникает, когда другое исключение возбуждается внутри блока `except`. Например:

```
>>> def example2():  
...     try:  
...         int('N/A')  
...     except ValueError as e:  
...         print("Couldn't parse:", err)  
...  
>>>  
>>> example2()  
Traceback (most recent call last):  
  File "<stdin>", line 3, in example2  
ValueError: invalid literal for int() with base 10: 'N/A'
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example2
    NameError: global name 'err' is not defined
>>>
```

В этом примере вы получаете информацию об обоих исключениях, но интерпретация немного отличается. В этом случае исключение *NameError* возбуждается как результат ошибки программирования, а не как прямой ответ на ошибку парсинга. Для этого случая атрибут исключения *__cause__* не установлен. Вместо этого установлен на предыдущее исключение атрибут *__context__*.

Если по какой-то причине вы хотите подавить образование цепочек, используйте *raise from None*:

```
>>> def example3():
...     try:
...         int('N/A')
...     except ValueError:
...         raise RuntimeError('A parsing error occurred') from None...
>>>
example3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example3
RuntimeError: A parsing error occurred
>>>
```

Обсуждение

При проектировании кода вы должны внимательно следить за использованием инструкции *raise* в других блоках *except*. В большинстве случаев такие инструкции *raise* должны быть, вероятно, изменены на *raise from*. Так что вы должны выбирать такой стиль:

```
try:
...
except SomeException as e:
    raise DifferentException() from e
```

Причина делать так в том, что вы явно связываете цепочку причин. *DifferentException* возбуждается как прямой ответ на получение *SomeException*. Это отношение будет явно показано в получившемся

трейсбэке.

Если вы пишете ваш код в нижеприведённом стиле, вы всё еще получите связанные в цепочку исключения, но часто неясно, были ли они объединены в цепочку намеренно или в результате непредвиденной программной ошибки:

```
try:  
    ...  
except SomeException:  
    raise DifferentException()
```

Когда вы используете *raise from*, вы четко показываете, что намеревались возбудить второе исключение.

Сопротивляйтесь желанию подавить информацию об исключении, как то показано в последнем примере. Хотя подавление информации об исключении может привести к меньшему размеру трейсбэка, это также удаляет информацию, которая могла бы быть полезна для отладки. При прочих равных часто лучше сохранить столько сведений, сколько возможно.

14.10. Повторное возбуждение последнего исключения

Задача

Вы поймали исключение в блоке `except`, но хотите заново возбудить его.

Решение

Просто используйте инструкцию `raise` саму по себе. Например:

```
>>> def example():  
...     try:  
...         int('N/A')  
...     except ValueError:  
...         print("Didn't work")  
...         raise  
...  
>>> example()  
Didn't work  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>
```

```
File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

Обсуждение

Эта задача часто возникает, когда вам нужно предпринять какое-то действие в ответ на исключение (например, логирование, очистка и т.п.), но после вы хотите продолжить распространение исключения. Очень распространённый пример использования — в улавливающих все исключения обработчиках:

```
try:
    ...
except Exception as e:
    # Как-то обработать информацию об исключении
    ...
    # Продолжить распространение исключения
    raise
```

14.11. Вывод предупреждающих сообщений

Задача

Вы хотите заставить вашу программу выдавать предупреждающие сообщения (например, об устаревших возможностях или проблемах с использованием).

Решение

Чтобы заставить вашу программу выдать предупреждающее сообщение, используйте функцию `warnings.warn()`. Например:

```
import warnings

def func(x, y, logfile=None, debug=False):
    if logfile is not None:
        warnings.warn('logfile argument deprecated', DeprecationWarning)
    ...

```

Аргументы для `warn()` — это предупреждающее сообщение вместе с классом предупреждения, в типичном случае — одним из следующих: `UserWarning`, `DeprecationWarning`, `SyntaxWarning`, `RuntimeWarning`, `ResourceWarning` или `FutureWarning`.

Обработка предупреждений зависит от того, как вы выполняете интерпретатор и прочей конфигурации. Например, если вы запускаете Python с опцией `-W all`, то получите такой вывод:

```
bash % python3 -W all example.py
example.py:5: DeprecationWarning: logfile argument is deprecated
    warnings.warn('logfile argument is deprecated', DeprecationWarning)
```

В обычном случае предупреждения просто выводят сообщения в стандартный поток ошибок. Если вы хотите превратить предупреждения в исключения, используйте опцию `-W error`:

```
bash % python3 -W error example.py
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    func(2, 3, logfile='log.txt')
  File "example.py", line 5, in func
    warnings.warn('logfile argument is deprecated', DeprecationWarning)
DeprecationWarning: logfile argument is deprecated
bash %
```

Обсуждение

Вывод предупреждающего сообщения часто бывает полезным приёмом для поддержки ПО и помочи пользователям с проблемами, которые необязательно поднимаются на уровень полноценного исключения.

Например, если вы собираетесь изменить поведение библиотеки или фреймворка, вы можете начать выдавать предупреждающие сообщения для частей, которые вы измените, при этом всё ещё предоставляя временную обратную совместимость. Вы также можете предупредить пользователей о проблемах при определённых случаях использования в их коде.

В качестве другого примера предупреждения во встроенной библиотеке можно привести сообщение, которое генерируется при уничтожении файла без закрытия:

```
>>> import warnings
```

```
>>> warnings.simplefilter('always')
>>> f = open('/etc/passwd')
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.TextIOWrapper name='/etc/
mode='r' encoding='UTF-8'>
>>>
```

По умолчанию показываются не все сообщения с предупреждениями. Опция запуска Python `-W` может контролировать вывод предупреждений. `-W all` выведет все предупреждения, `-W ignore` — проигнорирует все предупреждения, а `-W error` превратит сообщения об ошибках в исключения. В качестве альтернативы вы можете использовать функцию `warnings.simplefilter()`, чтобы контролировать вывод, как было показано выше. Аргумент `always` заставит появиться все предупреждения, `ignore` — проигнорировать все, `error` — превратить предупреждения в исключения.

Для простых случаев вам нужно просто выводить сообщения с предупреждениями. Модуль `warnings` также предоставляет разнообразные опции конфигурации, связанные с фильтрацией и обработкой предупреждающих сообщений. См. [документацию Python](#).

14.12. Отладка базовых падений программ

Задача

Ваша программа сломана, и вы хотите научиться простым стратегиям отладки.

Решение

Если ваша программа падает с исключением, полезно будет запустить её так: `python3 -i someprogram.py`. Опция `-i` запускает интерактивную оболочку, как только программа завершается. Из неё вы можете исследовать окружение. Например, у вас есть такой код:

```
# sample.py
def func(n):
    return n + 10
```

```
func('Hello')
```

Запуск `python3 -i` производит такой вывод:

```
bash % python3 -i sample.py
Traceback (most recent call last):
  File "sample.py", line 6, in <module>
    func('Hello')
  File "sample.py", line 4, in func
    return n + 10
TypeError: Can't convert 'int' object to str implicitly
>>> func(10)
20
>>>
```

Если вы не видите ничего очевидного, следующим шагом будет запуск дебаггера после падения. Например:

```
>>> import pdb
>>> pdb.pm()
> sample.py(4)func()
-> return n + 10
(Pdb) w
  sample.py(6)<module>()
-> func('Hello')
> sample.py(4)func()
-> return n + 10
(Pdb) print n
'Hello'
(Pdb) q
>>>
```

Если ваш код глубоко зарыт в окружение, где трудно получить интерактивную оболочку (например, на сервере), вы часто можете выловить ошибки и вывести трейсбэки самостоятельно. Например:

```
import traceback
import sys

try:
    func(arg)
except:
    print('***** AN ERROR OCCURRED *****')
    traceback.print_exc(file=sys.stderr)
```

Если ваша программа не падает, но выводит неправильные ответы, или вы

озадачены тем, как она работает, не будет неправильным решением добавить несколько вызовов `print()` в интересующих местах. Но если вы собираетесь это сделать, есть несколько похожих приёмов. Во-первых, функция `traceback.print_stack()` создаст вывод стека вашей программы прямо в этой точке. Например:

```
>>> def sample(n):
...     if n > 0:
...         sample(n-1)
...     else:
...         traceback.print_stack(file=sys.stderr)
...
>>> sample(5)
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in sample
File "<stdin>", line 5, in sample
>>>
```

В качестве альтернативы вы можете вручную запустить дебаггер в любой точке вашей программы с помощью `pdb.set_trace()`:

```
import pdb

def func(arg):
    ...
    pdb.set_trace()
    ...
```

Это может быть полезным приёмом для тыкания палочкой во внутренности большой программы и получения ответов на вопросы о потоке управления или аргументах функций. Например, после запуска дебаггера вы можете инспектировать переменные, используя `print`, или ввести команду типа `w`, чтобы получить трейсбэк стека.

Обсуждение

Не усложняйте отладку. Простые ошибки часто могут быть исправлены путём элементарных навыков чтения трейсбэков (ошибка обычно выводится в последней строчке трейсбэка). Отлично работает и помещение нескольких инструкций `print` в ваш код, если вы хотите получить какую-то

диагностическую информацию (просто не забудьте удалить их позже).

Обычно дебаггер используется для изучения переменных внутри сломавшейся функции. Умение войти в дебаггер после такого краха — это полезный навык.

Вставлять инструкции типа `pdb.set_trace()` — это полезно, если вы хотите распутать клубок очень сложной программы, поток управления которой неочевиден. По сути, программа будет выполняться до тех пор, пока не дойдёт до вызова `set_trace()`, и в этой точке сразу же зайдёт в дебаггер. Здесь вы уже можете попробовать разобраться в коде.

Если вы используете IDE для разработки на Python, она обычно предоставляет собственный интерфейс для отладки на базе `pdb` (или какой-то собственный). Обратитесь к руководству по вашей IDE за подробностями.

14.13. Профилирование и замеры времени выполнения вашей программы

Задача

Вы хотите узнать, на что ваша программа тратит время при выполнении, и сделать некоторые измерения.

Решение

Если вы просто хотите измерить время выполнения всей программы, обычно проще всего использовать что-то типа команды Unix `time`. Например:

```
bash % time python3 someprogram.py
real 0m13.937s
user 0m12.162s
sys 0m0.098s
bash %
```

Если же вы, наоборот, хотите получить детализированный отчёт о работе вашей программы, используйте модуль `cProfile`:

```
bash % python3 -m cProfile someprogram.py
859647 function calls in 16.016 CPU seconds
```

```

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall   filename:lineno(function)
263169    0.080    0.000    0.080    0.000  someprogram.py:16(franc
      513    0.001    0.000    0.002    0.000  someprogram.py:30(genera
262656    0.194    0.000   15.295    0.000  someprogram.py:32(<gen>
      1    0.036    0.036   16.077   16.077  someprogram.py:4(<module>
262144   15.021    0.000   15.021    0.000  someprogram.py:4(in_main
      1    0.000    0.000    0.000    0.000  os.py:746(urandom)
      1    0.000    0.000    0.000    0.000  png.py:1056(_readable)
      1    0.000    0.000    0.000    0.000  png.py:1073(Reader)
      1    0.227    0.227    0.438    0.438  png.py:163(<module>)
    512    0.010    0.000    0.010    0.000  png.py:200(group)

...
bash %

```

В большинстве случаев профилирование кода располагается где-то между этими двумя крайними точками. Например, вы уже можете знать, что ваш код тратит больше всего времени на несколько конкретных функций. Для выборочного профилирования функций будет полезен короткий декоратор. Например:

```

# timethis.py

import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        r = func(*args, **kwargs)
        end = time.perf_counter()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper

```

Чтобы использовать этот декоратор, вы просто помещаете его перед определением функции, для которой нужно получить тайминги. Например:

```

>>> @timethis
... def countdown(n):
...     while n > 0:
...         n -= 1
...
>>> countdown(10000000)

```

```
__main__.countdown : 0.803001880645752
>>>
```

Чтобы подсчитать время выполнения блока инструкций, вы можете определить менеджер контекста:

```
from contextlib import contextmanager

@contextmanager
def timeblock(label):
    start = time.perf_counter()
    try:
        yield
    finally:
        end = time.perf_counter()
        print('{} : {}'.format(label, end - start))
```

Вот пример его работы:

```
>>> with timeblock('counting'):
...     n = 10000000
...     while n > 0:
...         n -= 1
...
counting : 1.5551159381866455
>>>
```

Для изучения производительности небольших фрагментов кода будет полезен модуль *timeit*. Например:

```
>>> from timeit import timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>
```

timeit работает путём выполнения миллион раз инструкции, указанной в первом аргументе, и измерения потраченного на это времени. Второй аргумент — это строка с параметрами, которая выполняется для настройки окружения перед запуском теста. Если вы хотите изменить количество итераций, передайте аргумент *number*:

```
>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
```

```
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>
```

Обсуждение

При проведении измерений производительности нужно помнить, что любые результаты будут приблизительными. Функция `time.perf_counter()`, использованная в решении, предоставляет наиболее точный таймер из доступных на конкретной платформе. Однако она всё-таки измеряет внешнее время, и на результаты влияют различные факторы, такие как нагрузженность компьютера.

Если вы хотите получить время обработки, а не внешнее время, используйте `time.process_time()`. Например:

```
from functools import wraps
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.process_time()
        r = func(*args, **kwargs)
        end = time.process_time()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper
```

И последнее: если собираетесь проводить детальный анализ тайминга, ознакомьтесь с документацией `time`, `timeit` и других связанных модулей, а также разберитесь с различиями платформ и другими подводными камнями.

См. рецепт 13.13., где обсуждается создание класса-секундомера.

14.14. Заставляем ваши программы выполнять быстрее

Задача

Ваша программа работает слишком медленно, и вы хотели бы ускорить её без помощи сложных решений типа написания расширений на С или JIT-

компилиатора.

Решение

Первым правилом оптимизации могло бы стать «не делайте этого», но второе — это практически всегда «не оптимизируйте то, что не играет роли». Если ваша программа работает медленно, начните с профилирования кода, используя рецепт **14.13**.

В большинстве случаев вы обнаружите, что ваша программа тратит почти всё время в нескольких «горячих точках», таких как внутренние циклы обработки данных. Когда вы определили эти точки, вы можете использовать представленные ниже небезынтересные приёмы, чтобы заставить программу работать быстрее.

Используйте функции Многие программисты начинают использовать Python для написания простых скриптов. При этом легко приобрести привычку писать слабо структурированный код. Например:

```
# somescript.py

import sys
import csv

with open(sys.argv[1]) as f:
    for row in csv.reader(f):
        # Какая-то обработка
        ...
```

Малоизвестный факт: код, определённый в глобальной области видимости, как показано выше, выполняется медленнее, чем определённый в функции. Разница в скорости связана с реализацией локальных и глобальных переменных (операции с локальными выполняются быстрее). Так что если вы хотите заставить программу выполнять быстрее, просто поместите инструкции скрипта в функцию:

```
# somescript.py

import sys
import csv

def main(filename):
    with open(filename) as f:
```

```
for row in csv.reader(f):
    # Какая-то обработка
    ...

main(sys.argv[1])
```

Разница в скорости сильно зависит от выполняемой обработки, но, согласно нашему опыту, выигрыш в 15-30% не является чем-то необычным.

Выборочно удалите доступ к атрибутам Каждое использование оператора точки (.) для доступа к атрибутам имеет цену. «Под капотом» это задействует специальные методы, такие как `__getattribute__()` и `__getattr__()`, что часто приводит к поиску в словаре.

Вы можете избежать поиска атрибутов путём использования формы импортирования `from module import name`, а также выборочного использования связанных методов. Чтобы проиллюстрировать это, рассмотрим следующий фрагмент кода:

```
import math

def compute_roots(nums):
    result = []
    for n in nums:
        result.append(math.sqrt(n))
    return result

# Проверка
nums = range(1000000)
for n in range(100):
    r = compute_roots(nums)
```

При проверке на нашем компьютере, эта программа выполнялась около 40 секунд. Теперь изменим функцию `compute_roots()` следующим образом:

```
from math import sqrt

def compute_roots(nums):
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

Эта версия выполнилась примерно за 29 секунд. Единственная разница

между двумя версиями кода — отказ от доступа к атрибуту. Вместо `math.sqrt()` мы применили `sqrt()`. Дополнительно метод `result.append()` сохранён в локальную переменную `result_append` и повторно используется во внутреннем цикле.

Однако нужно подчеркнуть, что эти изменения имеют смысл только в часто исполняемом коде, таком как циклы. Подобная оптимизация имеет смысл только в отдельных местах.

Понимайте локальность переменных Как мы отметили выше, локальные переменные быстрее глобальных. Для имён, к которым часто осуществляется доступ, можно получить выигрыш в скорости, сделав максимальное количество этих имён локальными. Например, рассмотрите изменённую версию обсуждавшейся выше функции `compute_roots()`:

```
import math

def compute_roots(nums):
    sqrt = math.sqrt
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result
```

В этой версии `sqrt` вынут из модуля `math` и помещён в локальную переменную. Когда мы запустили этот код, то он выполнился примерно за 25 секунд (это улучшение по сравнению с предыдущей версией, выполнившейся за 29 секунд). Дополнительный прирост производительности достигнут благодаря тому, что локальный поиск `sqrt` немного быстрее глобального поиска `sqrt`.

Локальность аргументов также применяется при работе с классами. В общем случае поиск значения типа `self.name` будет заметно медленнее, чем доступ к локальной переменной. Во внутреннем цикле может окупиться перемещение атрибутов, к которым часто осуществляется доступ, в локальные переменные. Например:

```
# Медленно
class SomeClass:
    ...
    def method(self):
        for x in s:
```

```
op(self.value)

# Быстро
class SomeClass:
    ...
    def method(self):
        value = self.value
        for x in s:
            op(value)
```

Избегайте беспричинной абстракции Каждый раз, когда вы оборачиваете код дополнительными слоями обработки, такими как декораторы, свойства или дескрипторы, вы замедляете его. Например, рассмотрим такой класс:

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, value):
        self._y = value
```

Теперь проведём простой тест с таймингом:

```
>>> from timeit import timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

Как вы можете заметить, доступ к свойству у не просто медленнее доступа к простому атрибуту *x* — он примерно в 4,5 раза медленнее. Если эти разница важна, вы должны спросить себя, так ли необходимо определять *y* как свойство. Если нет, просто избавьтесь от него и вернитесь к использованию простого атрибута. Не стоит тащить в Python стиль программирования на базе геттеров и сеттеров только потому, что он принят в других языках.

Используйте встроенные контейнеры Встроенные типы данных, такие как строки, кортежи, списки, множества и словари, реализованы на С. Если вы склоняетесь к созданию собственной структуры данных (например, связного

списка, сбалансированного дерева и т.п.), вам будет сложно или даже невозможно сравниваться по скорости со встроенными типами. Так что обычно лучше использовать именно их.

Избегайте создания ненужных структур данных или копий Иногда программистов заносит, и они создают ненужные структуры данных, когда они просто-напросто не должны этого делать. Например, кто-то может написать такой код:

```
values = [x for x in sequence]
squares = [x*x for x in values]
```

Вероятно, у программиста была идея сначала собрать значения в список, а затем начать применять к ним операции типа генератора списка. Однако первая строчка не нужна. Просто напишите такой код:

```
squares = [x*x for x in sequence]
```

Будьте настороже и обращайте внимание на код, который пишется программистами, чрезмерно параноидально относящимися к разделению значений в Python. Излишнее использование функций типа `copy.deepcopy()` может быть признаком кода, который написан кем-то, кто не до конца понимает или доверяет модели памяти Python. В таком коде, вероятно, можно без опасений удалить множество копий.

Обсуждение

Перед оптимизированием обычно стоит изучить алгоритмы, которые вы используете. Вы получите намного более серьёзный выигрыш по скорости, переключившись на алгоритм со сложностью $O(n \log n)$, чем оптимизируя реализацию алгоритма $O(n^{**}2)$.

Если вы всё ещё полны решимости заняться оптимизацией, стоит взглянуть на общую картину. В качестве общего правила: не стоит оптимизировать все части программы, поскольку такие изменения делают код трудным для чтения и понимания. Лучше сфокусируйтесь на известных «бутылочных горлышках» производительности, таких как внутренние циклы.

Стоит осторожно интерпретировать результаты микрооптимизаций. Например, рассмотрим два приёма создания словаря:

```
a = {  
    'name' : 'AAPL',  
    'shares' : 100,  
    'price' : 534.22  
}  
  
b = dict(name='AAPL', shares=100, price=534.22)
```

В последнем случае вы получаете преимущество сокращения ввода с клавиатуры (не нужно закавычивать ключи). Однако если вы выставите эти фрагменты кода на состязание, то обнаружите, что `dict()` работает втрое медленнее! Вооружившись этим знанием, вы можете склониться к просмотру всего кода и замене каждого вызова `dict()` более многословной альтернативой. Однако умный программист сфокусируется только на частях программы, где это действительно важно — таких как внутренние циклы. В других местах разница в скорости не будет играть роли.

Если же вопросы производительности в вашем случае требуют идти намного дальше представленных в этом рецепте простых приёмов, вам стоит изучить инструменты на базе JIT-компиляторов (*just-in-time*). Например, [проект PyPy](#) — это альтернативная реализация интерпретатора Python, которая анализирует выполнение вашей программы и генерирует нативный машинный код для часто выполняющихся частей. Иногда с его помощью можно заставить программы выполнять на порядок быстрее, часто приближаясь по скорости к коду, написанному на С. К сожалению, на момент написания этой книги PyPy ещё не полностью поддерживает Python 3 (прим. пер.: в 2015-м с поддержкой Python 3 всё намного лучше). Вы также можете посмотреть на проект [Numba](#). Это динамический компилятор, который оптимизирует выбранные вами (аннотированные декоратором) функции Python. Эти функции затем компилируются в нативный машинный код с помощью [LLVM](#). Это также может привести к значительному приросту скорости. Однако, как и в случае PyPy, поддержка Python 3 здесь экспериментальная (прим. пер.: в 2015-м году уже заявлена официальная поддержка).

Напоследок вспомним слова Джона Остерхута: «Лучшее улучшение производительности — это переход от нерабочего к рабочему состоянию». Не волнуйтесь об оптимизации до тех пор, пока вам это не потребуется. Убедитесь, что ваша программа работает правильно — обычно это важнее скорости (по крайней мере, на начальном этапе).

15. Расширения на языке С

Эта глава рассматривает проблему доступа к коду на С из Python. Многие встроенные библиотеки Python написаны на С, и доступ к С — это важная часть налаживания работы Python с существующими библиотеками. Это также область, которая может потребовать наибольших усилий для понимания, если вы столкнулись с задачей переноса существующего кода с Python 2 на Python 3.

Хотя Python предоставляет обширный API для программирования на С, существует много подходов для работы с этим языком. Вместо того, чтобы попытаться дать здесь выматывающий компендиум всех возможных инструментов или приёмов, мы сфокусируемся на небольшом фрагменте кода на С вместе с несколькими репрезентативными примерами работы с ним. Цель — предоставить набор шаблонов программирования, который опытные программисты могут расширить для собственного использования.

Вот код на С, который будет работать в большинстве рецептов:

```
/* sample.c */_method
#include <math.h>

/* Вычисляет наибольший общий делитель */
int gcd(int x, int y) {
    int g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}

/* Проверяет, является ли (x0,y0) множеством Мандельброта */
int in_mandel(double x0, double y0, int n) {
    double x=0,y=0,xtemp;
    while (n > 0) {
        xtemp = x*x - y*y + x0;
        y = 2*x*y + y0;
        x = xtemp;
        n -= 1;
        if (x*x + y*y > 4) return 0;
    }
    return 1;
}
```

```

/* Делит одно число на другое */
int divide(int a, int b, int *remainder) {
    int quot = a / b;
    *remainder = a % b;
    return quot;
}

/* Усредняет значения массива */
double avg(double *a, int n) {
    int i;
    double total = 0.0;
    for (i = 0; i < n; i++) {
        total += a[i];
    }
    return total / n;
}

/* Структура данных C */
typedef struct Point {
    double x,y;
} Point;

/* Функция с использованием структуры данных C */
double distance(Point *p1, Point *p2) {
    return hypot(p1->x - p2->x, p1->y - p2->y);
}

```

Этот код содержит несколько разных реализованных на С возможностей. Во-первых, здесь есть несколько простых функций, таких как `gcd()` и `is_mandel()`. Функция `divide()` — это пример функции С, возвращающей несколько значений, каждое через аргумент-указатель. Функция `avg()` выполняет сокращение (свёртку) данных в массиве С. `Point` и функция `distance()` используют структуры С.

Во всех последующих рецептах мы предполагаем, что приведённый выше код находится в файле с именем `sample.c`, чьи определения находятся в файле `sample.h`, и он скомпилирован в библиотеку `libsample`, которая может быть подлинкована к другому коду на С. Конкретные детали компилирования и линковки варьируются от системы к системе, но мы не будем на этом фокусироваться. Мы предполагаем, что если вы работаете с кодом на С, то вы уже в этом разобрались.

15.1. Доступ к коду на С с использованием ctypes

Задача

У вас есть небольшое количество функций на С, которые были скомпилированы в разделяемую библиотеку или DLL. Вы хотели бы вызвать эти функции просто из Python, без необходимости писать дополнительный код на С (и без использования стороннего инструмента для расширений).

Решение

Для небольших задач, использующих код на С, часто достаточно просто использовать модуль `ctypes`, который является частью стандартной библиотеки Python. Чтобы использовать `ctypes`, сначала вы должны убедиться, что код на С, к которому вы хотите получить доступ, был скомпилирован в разделяемую библиотеку, совместимую с интерпретатором Python (то есть он использует ту же архитектуру, размер слова, компилятор и т.п.) Для целей этого рецепта мы предположим, что разделяемая библиотека `libsample.so` создана и не содержит ничего, кроме кода, показанного во введении в главу. Также мы предположим, что `libsample.so` размещена в том же каталоге, что и рассмотренный ниже файл `sample.py`.

Чтобы обратиться к получившейся библиотеке, вы создаете модуль Python, который обворачивает её, как показано тут:

```
# sample.py
import ctypes
import os

# Пытаемся найти .so-файл в том же каталоге, что и этот файл
_file = 'libsample.so'
_path = os.path.join(*(os.path.split(__file__)[-1] + (_file,)))
_mod = ctypes.cdll.LoadLibrary(_path)

# int gcd(int, int)
gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int

# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int

# int divide(int, int, int *)
divide = _mod.divide
divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.c_int)
divide.restype = ctypes.c_int
```

```

_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x, y, rem)
    return quot,rem.value

# void avg(double *, int n)
# Определяет специальный тип для аргумента 'double *'
class DoubleArrayType:
    def from_param(self, param):
        typename = type(param).__name__
        if hasattr(self, 'from_' + typename):
            return getattr(self, 'from_' + typename)(param)
        elif isinstance(param, ctypes.Array):
            return param
        else:
            raise TypeError("Can't convert %s" % typename)

# Перекладовывает (cast) из объектов array.array
def from_array(self, param):
    if param.typecode != 'd':
        raise TypeError('must be an array of doubles')
    ptr, _ = param.buffer_info()
    return ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))

# Перекладовывает (cast) из списков/кортежей
def from_list(self, param):
    val = ((ctypes.c_double)*len(param))(*param)
    return val

from_tuple = from_list

# Перекладовывает (cast) из массивов numpy
def from_ndarray(self, param):
    return param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))

DoubleArray = DoubleArrayType()
_avg = _mod.avg
_avg.argtypes = (DoubleArray, ctypes.c_int)
_avg.restype = ctypes.c_double

def avg(values):
    return _avg(values, len(values))

# struct Point { }
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

```

```
# double distance(Point *, Point *)
distance = _mod.distance
distance.argtypes = (ctypes.POINTER(Point), ctypes.POINTER(Point))
distance.restype = ctypes.c_double
```

Если всё пойдёт хорошо, то вы сможете загрузить модуль и использовать получившиеся функции С. Например:

```
>>> import sample
>>> sample.gcd(35,42)
7
>>> sample.in_mandel(0,0,500)
1
>>> sample.in_mandel(2.0,1.0,500)
0
>>> sample.divide(42,8)
(5, 2)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

Обсуждение

В этом рецепте несколько аспектов, которые требуют некоторого обсуждения. Первый момент касается упаковки кода Python и С воедино. Если вы используете `ctypes` для доступа к коду на С, который вы скомпилировали сами, то вам нужно убедиться, что разделяемая библиотека размещена в месте, где модуль `sample.py` сможет её найти. Один путь — поместить получившийся .so-файл в тот же каталог, что и поддерживающий код на Python. Это показано в первой части рецепта — `sample.py` ищет переменную `_file_`, чтобы узнать, где она установлена, а затем конструирует путь, который указывает на файл `libsample.so` в том же каталоге.

Если библиотека на С будет инсталлирована где-то ещё, то вы должны соответствующим образом подправить путь. Если библиотека на С установлена в качестве стандартной библиотеки на вашем компьютере, вы можете использовать функцию `ctypes.util.find_library()`. Например:

```
>>> from ctypes.util import find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
>>>
```

Ещё раз: *ctypes* не будет работать, если он не сможет обнаружить библиотеку с кодом на С. Так что вам нужно провести несколько минут и подумать о том, как вы хотите всё инсталлировать.

Когда вы узнали, где размещена библиотека на С, вы используете *ctypes.cdll.LoadLibrary()*, чтобы её загрузить. Следующая инструкция в решении делает это (*_path* — это полный путь к разделяемой библиотеке):

```
_mod = ctypes.cdll.LoadLibrary(_path)
```

Когда библиотека загружена, вам нужно написать инструкции, которые извлекают конкретные символы и помещают на них сигнатуры типов. Это случится с подобными фрагментами кода:

```
# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int
```

В этом коде атрибут *.argtypes* — это кортеж, который содержит входные аргументы функции, а *.restype* — это возвращаемый тип. *ctypes* определяет различные объекты типов (например, *c_double*, *c_int*, *c_short*, *c_float* и т.д.), которые представляют часто используемые типы данных С. Прикрепление сигнатур типов крайне важно, если вы хотите заставить Python передавать правильные типы аргументов и правильно преобразовывать данные (если вы не сделаете этого, код не только не будет работать, но и может вызвать падение всего интерпретатора).

Тонкий момент использования *ctypes* в том, что изначальный код на С может использовать идиомы, которые не отображаются однозначно на Python. Функция *divide()* — хороший пример, поскольку она возвращает значение через один из своих аргументов. Хотя в С это распространённый приём, часто не совсем понятно, как это должно работать в Python. Например, вы не

можете сделать это напрямую:

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected LP_c_int
instance instead of int
>>>
```

Даже если бы это сработало, это бы нарушило неизменяемость целых чисел в Python и, вероятно, вызывало бы засасывание всего интерпретатора в чёрную дыру. Для аргументов, использующих указатели, вам обычно приходится конструировать совместимый объект *ctypes* и передавать его таким образом:

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
>>>
```

Здесь экземпляр *ctypes.c_int* создан и передан как объект-указатель. В отличие от обычного целого числа Python, объект *c_int* может быть изменён. Атрибут *.value* может быть использован по желанию либо для получения, либо для изменения значения.

Для классов, где соглашения о вызове С «непитоничны», часто пишут маленькую функцию-обёртку. В решении этот код заставляет функцию *divide()* вернуть два результата в кортеже:

```
# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x,y,rem)
    return quot, rem.value
```

Функция `avg()` — это ещё один вызов мастерству программиста. Лежащий в основе код на С ожидает получить указатель и длину, представляющую массив. Однако со стороны Python вы должны задаться следующими вопросами: «Что такое массив? Это список? Это кортеж? Массив из модуля `array`? Массив `ptr`? Или всё вместе взятое?» На практике массив Python может принимать множество различных форм, и, возможно, вы захотите поддержать несколько возможных вариантов.

Класс `DoubleArrayType` показывает, как справиться с этой ситуацией. В этом классе определён единственный метод `from_param()`. Роль этого метода — принять единственный параметр и сузить его до совместимого объекта `ctypes` (например, указатель до `ctypes.c_double`). Внутри `from_param()` вы свободны делать всё, что пожелаете. В решении из параметра извлекается имя типа и используется для диспетчеризации в более специализированный метод. Например, если передан список, имя типа — `list`, что вызывает метод `from_list()`.

Для списков и кортежей метод `from_list()` выполняет преобразование в объект массива `ctypes`. Это выглядит немного странно, но вот интерактивный пример преобразования списка в массив `ctypes`:

```
>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>
```

Для объектов `array` метод `from_array()` извлекает лежащий в основе указатель на память и «переколдовывает» (приводит, `cast`) его в объект указателя `ctypes`. Например:

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr_ = a.buffer_info()
>>> ptr
4298687200
```

```
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
>>>
```

`from_ndarray()` демонстрирует код для совместимого преобразования массивов *numpy*.

Путём определения класса *DoubleArrayType* и использования его в сигнатуре типа функции *avg()*, как тут показано, функция может принимать разнообразные массивоподобные входные аргументы:

```
>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
```

Последняя часть этого рецепта показывает, как работать с простыми структурами С. Для структур вы просто определяете класс, который содержит подходящие поля и типы:

```
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]
```

Когда класс определён, вы можете использовать его в сигнтурах типов, а также в коде, где нужно создать экземпляр структуры и работать с ним. Например:

```
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

Несколько последних комментариев: *ctypes* — это полезная библиотека, о которой нужно знать, если вам просто нужно получить доступ к некоторым функциям на С из Python. Однако если вы пытаетесь обратиться к крупной библиотеке, то вам стоит попробовать другие подходы, такие как Swig (описан в рецепте 15.9.) или Cython (описан в рецепте 15.10.)

Главная проблема с крупными библиотеками в том, что *ctypes* не является полностью автоматической: вам нужно провести немало времени, прописывая все сигнатуры типов, как показано в примере. В зависимости от сложности библиотеки, вам также, возможно, придётся написать множество небольших функций-обёрток и поддерживающих классов. Также, если вы досконально не разбираетесь в низкоуровневых деталях интерфейса С, включая управление памятью и обработку ошибок, часто очень легко обрушить Python с ошибкой сегментации (segmentation fault), нарушением доступа (access violation) или другой подобной ошибкой.

В качестве альтернативы *ctypes* вы можете также взглянуть на [CFFI](#). CFFI предоставляет практически такую же функциональность, но использует синтаксис С и поддерживает более продвинутые типы кода на С. На момент написания этой книги CFFI был молодым проектом, но быстро рос. Обсуждается также возможность включения проекта в стандартную библиотеку Python в одном из будущих релизов. Так что его стоит не упускать из виду.

15.2. Написание простого модуля расширения на С

Задача

Вы хотите написать простой модуль расширения на С, напрямую используя API расширений Python. Вы не хотите использовать никакие другие инструменты.

Решение

Для простого кода на С сделать собственный модуль расширения довольно просто. В качестве подготовительного шага вы, вероятно, захотите убедиться, что ваш код на С имеет правильный заголовочный файл.

Например:

```
/* sample.h */

#include <math.h>

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

В типичном случае этот заголовок будет соответствовать библиотеке, которая была скомпилирована отдельно. Предполагая это, взгляните на пример модуля расширения, который иллюстрирует базовые принципы написания функций расширений:

```
#include "Python.h"
#include "sample.h"

/* int gcd(int, int) */
static PyObject *py_gcd(PyObject *self, PyObject *args) {
    int x, y, result;

    if (!PyArg_ParseTuple(args, "ii", &x, &y)) {
        return NULL;
    }
    result = gcd(x,y);
    return Py_BuildValue("i", result);
}

/* int in_mandel(double, double, int) */
static PyObject *py_in_mandel(PyObject *self, PyObject *args) {
    double x0, y0;
    int n;
    int result;

    if (!PyArg_ParseTuple(args, "ddi", &x0, &y0, &n)) {
        return NULL;
    }
    result = in_mandel(x0,y0,n);
    return Py_BuildValue("i", result);
}
```

```

/* int divide(int, int, int *) */
static PyObject *py_divide(PyObject *self, PyObject *args) {
    int a, b, quotient, remainder;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    quotient = divide(a,b, &remainder);
    return Py_BuildValue("(ii)", quotient, remainder);
}

/* Таблица методов модуля */
static PyMethodDef SampleMethods[] = {
    {"gcd", py_gcd, METH_VARARGS, "Greatest common divisor"},  

    {"in_mandel", py_in_mandel, METH_VARARGS, "Mandelbrot test"},  

    {"divide", py_divide, METH_VARARGS, "Integer division"},  

    {NULL, NULL, 0, NULL}
};

/* Структура модуля */
static struct PyModuleDef samplemodule = {
    PyModuleDef_HEAD_INIT,  

    "sample", /* Имя модуля */  

    "A sample module", /* Стока документирования (может быть NULL) */  

    -1, /* Размер состояния на каждый интерпретатор или  

    SampleMethods /* Таблица методов */
};

/* Функция инициализации модуля */
PyMODINIT_FUNC
PyInit_sample(void) {
    return PyModule_Create(&samplemodule);
}

```

Для компиляции модуля расширения создайте файл `setup.py`, который выглядит так:

```

# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      ext_modules=[  

          Extension('sample',  

                  ['pysample.c'],  

                  include_dirs = ['/some/dir'],  

                  define_macros = [('FOO','1')],  

                  undef_macros = ['BAR'],  

                  library_dirs = ['/usr/local/lib'],  

                  libraries = ['sample'])

```

```
)  
]  
)
```

Теперь, чтобы скомпилировать получившуюся библиотеку, просто используйте `python3 buildlib.py build_ext --inplace`:

```
bash % python3 setup.py build_ext --inplace  
running build_ext  
building 'sample' extension  
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototyp  
-I/usr/local/include/python3.3m -c pysample.c  
-o build/temp.macosx-10.6-x86_64-3.3/pysample.o  
gcc -bundle -undefined dynamic_lookup  
build/temp.macosx-10.6-x86_64-3.3/pysample.o \  
-L/usr/local/lib -lsample -o sample.so  
bash %
```

Как тут показано, это создаст разделяемую библиотеку `sample.so`. Когда она скомпилируется, вы сможете импортировать её как модуль:

```
>>> import sample  
>>> sample.gcd(35, 42)  
7  
>>> sample.in_mandel(0, 0, 500)  
1  
>>> sample.in_mandel(2.0, 1.0, 500)  
0  
>>> sample.divide(42, 8)  
(5, 2)  
>>>
```

Если вы попытаетесь выполнить эти шаги на Windows, то вам потребуется покорпеть над вашим окружением, чтобы заставить окружение компиляции правильно собрать модули расширения. Бинарные установочные файлы Python обычно собираются с помощью Microsoft Visual Studio. Чтобы заставить расширения работать, вам может потребоваться скомпилировать их с помощью совместимых инструментов. См. [документацию Python](#).

Обсуждение

Перед тем, как создавать какие-то самописные расширения, крайне важно ознакомиться с разделом документации Python «[Расширение и встраивание в интерпретатор Python](#)». API Python для расширений на Python весьма

обширен, и повторять его здесь непрактично. Однако самые важные части мы обсудим.

Во-первых, в модулях расширения функции, которые вы пишете, в обычном случае написаны с помощью обычного прототипа, как показано тут:

```
static PyObject *py_func(PyObject *self, PyObject *args) {  
    ...  
}
```

PyObject — это тип данных С, который представляет объект Python. На очень высоком уровне функция расширения — это функция С, которая получает кортеж объектов Python (в **args PyObject*) и возвращает новый объект Python в качестве результата. Аргумент функции *self* не используется в простых функциях расширений, но входит в игру, если вы должны определить новые классы или типы объектов в С (например, если функция расширения является методом класса, то *self* будет содержать экземпляр).

Функция *PyArg_ParseTuple()* используется для преобразования значений из Python в представление С. На входе она получает строку формата, которая указывает требуемые значения, такие как “i” для целых чисел и “d” для чисел двойной точности, а также адреса переменных С, в которые нужно поместить преобразованные результаты. *PyArg_ParseTuple()* выполняет разнообразные проверки количества и типов аргументов. Если имеет место несовпадение со строкой формата, возбуждается исключение и возвращается NULL. Путём проверки на это и простого возвращения NULL, в вызывающем коде будет возбуждено соответствующее исключение.

Функция *Py_BuildValue()* используется для создания объектов Python из типов данных С. Она также принимает код формата, чтобы обозначить нужный тип. В функциях расширения это используется возвращения результатов обратно в Python. *Py_BuildValue()* может создавать более сложные объекты, такие как кортежи и словари. В коде *py_divide()* показан пример возвращения кортежа. Однако есть и другие примеры:

```
return Py_BuildValue("i", 34);  
// Возвращает целое число  
return Py_BuildValue("d", 3.4);  
// Возвращает double  
return Py_BuildValue("s", "Hello");  
// Нуль-терминированная строка в UTF-8  
return Py_BuildValue("(ii)", 3, 4);
```

```
// Кортеж (3, 4)
```

В конце любого модуля расширения вы обнаружите таблицу функций, похожую на *SampleMethods* из решения, показанного в этом рецепте. В этой таблице перечислены функции С, имена для использования в Python, а также строки документации. Все модули должны определять такую таблицу, поскольку она используется при инициализации модуля.

Последняя функция *PyInit_sample()* — это функция инициализации модуля, которая выполняется, когда модуль впервые импортируется. Основная работа этой функции — зарегистрировать объект модуля в интерпретаторе.

И последнее: нужно подчеркнуть, что о расширении Python функциями на С можно рассказать намного больше, чем показано тут (С API содержит более 500 функций). Вам стоит рассматривать этот рецепт как первую ступеньку в восхождении к освоению этой темы. Начните изучение с документации функций *PyArg_ParseTuple()* и *Py_BuildValue()* и расширяйтесь из этой точки.

15.3. Написание функции расширения для работы с массивами

Задача

Вы хотите написать на С функцию расширения, которая работает со смежными массивами данных, которые могут быть созданы с помощью модуля *array* или библиотек типа NumPy. Однако вы хотели бы, чтобы ваша функция имела широкое применение, а не была специфичной для одной библиотеки для работы с массивами.

Решение

Чтобы получать и обрабатывать массивы переносимым способом, вы должны писать код, который использует [буферный протокол](#). Вот пример самописной функции расширения на С, которая принимает массив данных и вызывает функцию *avg(double *buf, int len)* из кода, приведённого во введении в эту главу:

```
/* Вызов double avg(double *, int) */
static PyObject *py_avg(PyObject *self, PyObject *args) {
```

```

PyObject *bufobj;
Py_buffer view;
double result;
/* Get the passed Python object */
if (!PyArg_ParseTuple(args, "O", &bufobj)) {
    return NULL;
}

/* Попытка извлечь информацию из буфера */
if (PyObject_GetBuffer(bufobj, &view,
    PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    return NULL;
}

if (view.ndim != 1) {
    PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional array");
    PyBuffer_Release(&view);
    return NULL;
}

/* Проверка типа элементов в массиве */
if (strcmp(view.format, "d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    PyBuffer_Release(&view);
    return NULL;
}

/* Передает сырой буфер и размер в функцию на C */
result = avg(view.buf, view.shape[0]);

/* Сигнализирует, что мы закончили работать с буфером */
PyBuffer_Release(&view);
return Py_BuildValue("d", result);
}

```

Вот пример работы этой функции расширения:

```

>>> import array
>>> avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> avg(numpy.array([1.0,2.0,3.0]))
2.0
>>> avg([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' does not support the buffer interface
>>> avg(b'Hello')
Traceback (most recent call last):

```

```
File "<stdin>", line 1, in <module>
TypeError: Expected an array of doubles
>>> a = numpy.array([[1.,2.,3.],[4.,5.,6.]])
>>> avg(a[:,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ndarray is not contiguous
>>> sample.avg(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected a 1-dimensional array
>>> sample.avg(a[0])
2.0
>>>
```

Обсуждение

Передача объектов массивов в функции на C — одно из самых распространённых применений функций расширения. Многие приложения на Python, от программ для обработки изображений до научных вычислений, основаны на высокопроизводительной обработке массивов. Путём написания кода, который может принимать массивы и работать с ними, вы можете разработать собственную программу, которая хорошо работает с этими приложениями, а не какое-то кастомное решение, которое будет работать только с вашим кодом.

Основной момент этой программы — функция *PyBuffer_GetBuffer()*. Когда её передают произвольный объект Python, она пытается получить информацию о лежащем в основе представлении памяти. Если это невозможно, как это и есть в случае обычных объектов Python, она просто возбуждает исключение и возвращает -1. Специальные флаги, передаваемые *PyBuffer_GetBuffer()*, дают дополнительные подсказки о типе запрашиваемого буфера памяти. Например, *PyBUF_ANY_CONTIGUOUS* определяет, что запрашивается смежная область памяти.

Для массивов, байтовых строк и других подобных объектов структура *Py_buffer* наполняется информацией о лежащей в основе памяти. Это включает указатель на память, размер, размер элемента, формат и другие детали. Вот определение этой структуры:

```
typedef struct bufferinfo {
    void *buf;                      /* Указатель на память буфера */
    PyObject *obj;                  /* Объект Python, являющийся владельцем */
```

```
Py_ssize_t len;           /* Общий размер в байтах */
Py_ssize_t itemsize;      /* Размер одного элемента в байтах */
int readonly;             /* Флаг доступа только на чтение */
int ndim;                /* Количество измерений */
char *format;             /* Код struct одного элемента */
Py_ssize_t *shape;        /* Массив, содержащий измерения */
Py_ssize_t *strides;      /* Массив, содержащий размер шага */
Py_ssize_t *suboffsets;    /* Массив, содержащий подсдвиги (suboffsets) */
} Py_buffer;
```

В этом рецепте мы обеспечиваем получение смежного массива чисел с двойной точностью (*doubles*). Чтобы проверить, являются ли элементы такими числами, проверяется атрибут *format*: чтобы посмотреть, есть ли там строка "d". Это тот же код, который модуль *struct* использует при кодировании бинарных значений. В качестве общего правила: *format* может быть любой строкой формата, совместимой с модулем *struct*, и может включать несколько элементов в случае массивов, содержащих структуры С.

Когда мы проверили «подкапотную» информацию буфера, мы просто передаём его в функцию С, которая обращается с ним, как с обычным массивом С. Для всех практических применений не нужно думать о том, какого типа этот массив, или о том, какая библиотека его создала. Таким образом функция может работать с массивами, созданными модулем *array* или *pintu*.

Перед возвращением конечного результата лежащий в основе буфер должен быть освобождён с помощью *PyBuffer_Release()*. Этот шаг необходим для правильного управления счётчиками ссылок объектов.

Повторимся: этот рецепт показывает лишь небольшой фрагмент кода, принимающего массив. При работе с массивами вы можете столкнуться с проблемами, связанными с многомерными данными, размером шага данных, различными типами данных и другими вопросами, требующими изучения. Обратитесь к [официальной документации](#) за подробностями.

Если вам нужно написать много расширений, в которых используется работа с массивами, то вам, возможно, легче будет реализовать их на Cython. См. рецепт 15.11.

15.4. Управление непрозрачными указателями в модулях расширения на С

Задача

У вас есть модуль расширения, которому нужно работать с указателем на структуру данных C, но вы не хотите показывать коду на Python внутреннее устройство структуры.

Решение

С непрозрачными структурами данных легко работать путём обравчивания их в объекты-капсулы. Рассмотрим фрагмент кода на C из нашего примера:

```
typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

Вот пример кода расширения, который обравчивает структуру *Point* и функцию *distance()* с помощью капсул:

```
/* Функция-деструктор для точек */
static void del_Point(PyObject *obj) {
    free(PyCapsule_GetPointer(obj, "Point"));
}

/* Вспомогательные функции */
static Point *PyPoint_AsPoint(PyObject *obj) {
    return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
    return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}

/* Создаёт новый объект Point */
static PyObject *py_Point(PyObject *self, PyObject *args) {
    Point *p;
    double x,y;
    if (!PyArg_ParseTuple(args, "dd",&x,&y)) {
        return NULL;
    }
    p = (Point *) malloc(sizeof(Point));
    p->x = x;
    p->y = y;
    return PyPoint_FromPoint(p, 1);
```

```

}

static PyObject *py_distance(PyObject *self, PyObject *args) {
    Point *p1, *p2;
    PyObject *py_p1, *py_p2;
    double result;

    if (!PyArg_ParseTuple(args, "OO", &py_p1, &py_p2)) {
        return NULL;
    }
    if (!(p1 = PyPoint_AsPoint(py_p1))) {
        return NULL;
    }
    if (!(p2 = PyPoint_AsPoint(py_p2))) {
        return NULL;
    }
    result = distance(p1, p2);
    return Py_BuildValue("d", result);
}

```

Вот как использовать эти функции из Python:

```

>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1004ea330>
>>> p2
<capsule object "Point" at 0x1005d1db0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>

```

Обсуждение

Капсулы похожи на типизированный указатель С. Внутри они содержат общий (generic) указатель вместе с идентификационным именем, и могут быть легко созданы с помощью функции *PyCapsule_New()*. Также необязательная функция-деструктор может быть прикреплена к капсule для освобождения памяти, когда объект капсулы попадает под сборку мусора.

Чтобы извлечь указатель, содержащийся в капсule, используйте функцию *PyCapsule_GetPointer()* и укажите имя. Если предоставленное имя не совпадает с тем, что есть в капсule (или произошли какие-либо другие ошибки), возбуждается исключение и возвращается NULL.

В этом рецепте две вспомогательные функции — *PyPoint_FromPoint()* и *PyPoint_AsPoint()* — написаны, чтобы работать с механикой создания и закрытия экземпляров *Point* из объектов капсул. В любых функциях расширения мы будем использовать эти функции вместо работы с капсулами напрямую. Этот выбор был сделан при проектировании, чтобы облегчить внесение будущих изменений в обёртку объектов *Point*. Например, если вы решите использовать что-то другое взамен капсул, вам нужно будет изменить только эти две функции.

Тонкий момент использования капсул касается сборки мусора и управления памятью. Функция *PyPoint_FromPoint()* принимает аргумент *must_free*, который показывает, должна ли лежащая в основе структура *Point* * быть собрана (с точки зрения сборки мусора), когда капсула будет уничтожена. При работе с некоторыми типами кода на С может быть трудно разобраться с проблемами, связанными с владением (например, структура *Point* встроена в крупную структуру данных, которая управляет отдельно). Вместо принятия одностороннего решения о сборке мусора, этот дополнительный аргумент передаёт управление обратно программисту. Стоит отметить, что деструктор, ассоциированный с существующей капсулой, также может быть изменён с помощью функции *PyCapsule_SetDestructor()*.

Капсулы — это разумное решение для создания интерфейсов для некоторых типов кода на С, использующих структуры. Например, иногда вы просто не должны заботиться о показывании внутреннего устройства структуры или превращения её в полноценный тип расширения. При использовании капсул вы можете обернуть её легковесной обёрткой и без труда передать другим функциям расширения.

15.5. Определение и экспорт C API из модулей расширения

Задача

У вас есть модуль расширения на С, который внутри себя определяет разнообразные полезные функции, которые вы бы хотели экспортовать для использования в других местах как публичный С API. Вы хотели бы использовать эти функции в других модулях расширения, но не знаете, как связать их вместе, поскольку сделать это с помощью компилятора/

линковщика языка С кажется чрезмерно сложной или даже невозможной задачей.

Решение

Этот рецепт фокусируется на коде, написанном для работы с объектами *Point*, который был представлен в **рецепте 15.4**. Как вы помните, этот код на С включает некоторые полезные функции:

```
/* Функция-деструктор для точек */
static void del_Point(PyObject *obj) {
    free(PyCapsule_GetPointer(obj, "Point"));
}

/* Вспомогательные функции */
static Point *PyPoint_AsPoint(PyObject *obj) {
    return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
    return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}
```

Проблема, которую мы сейчас решаем, заключается в том, как экспортовать функции *Point_AsPoint()* и *PyPoint_FromPoint()* как API, который другие модули расширения могут использовать и прилинковываться к нему (например, если у вас есть другие расширения, которые тоже хотят использовать обёрнутые объекты *Point*).

Чтобы решить эту задачу, введём новый заголовочный файл для расширения-примера под названием *pysample.h*. Поместим в него следующий код:

```
/* pysample.h */
#include "Python.h"
#include "sample.h"
#ifndef __cplusplus
extern "C" {
#endif

/* Таблица публичного API */
typedef struct {
    Point *(*aspoint)(PyObject *);
    PyObject *(*frompoint)(Point *, int);
} _PointAPIMethods;
```

```

#ifndef PYSAMPLE_MODULE
/* Таблица методов во внешнем модуле */
static _PointAPIMethods *_point_api = 0;

/* Импортирование таблицы API из примера */
static int import_sample(void) {
    _point_api = (_PointAPIMethods *) PyCapsule_Import("sample._point_api");
    return (_point_api != NULL) ? 1 : 0;
}

/* Макросы для реализации программного интерфейса */
#define PyPoint_AsPoint(obj) (_point_api->aspoint)(obj)
#define PyPoint_FromPoint(obj) (_point_api->frompoint)(obj)
#endif

#ifdef __cplusplus
}
#endif

```

Главный аспект здесь — это таблица указателей функций `_PointAPIMethods`. Она будет инициализирована в экспортирующем модуле и обнаружена в импортирующих модулях.

Измените изначальный модуль расширения, чтобы наполнить таблицу и экспортировать её, как показано:

```

/* pysample.c */

#include "Python.h"
#define PYSAMPLE_MODULE
#include "pysample.h"

...
/* Функция-деструктор для точек */
static void del_Point(PyObject *obj) {
    printf("Deleting point\n");
    free(PyCapsule_GetPointer(obj, "Point"));
}

/* Вспомогательные функции */
static Point *PyPoint_AsPoint(PyObject *obj) {
    return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int free) {
    return PyCapsule_New(p, "Point", free ? del_Point : NULL);
}

```

```
static _PointAPIMethods _point_api = {
    PyPoint_AsPoint,
    PyPoint_FromPoint
};

...

/* Функция инициализации модуля */
PyMODINIT_FUNC
PyInit_sample(void) {
    PyObject *m;
    PyObject *py_point_api;

    m = PyModule_Create(&samplemodule);
    if (m == NULL)
        return NULL;

    /* Добавляет функции Point C API */
    py_point_api = PyCapsule_New((void *) &_point_api, "sample._point_api",
        if (py_point_api) {
            PyModule_AddObject(m, "_point_api", py_point_api);
        }
    return m;
}
```

Наконец, вот пример нового модуля расширения, который загружает и использует эти функции API:

```
/* ptexample.c */

/* Включаем заголовок, ассоциированный с другим модулем */
#include "pysample.h"

/* Существующая функция, которая использует экспортированный API */
static PyObject *print_point(PyObject *self, PyObject *args) {
    PyObject *obj;
    Point *p;
    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }

    /* Заметьте: это определено в другом модуле */
    p = PyPoint_AsPoint(obj);
    if (!p) {
        return NULL;
    }
    printf("%f %f\n", p->x, p->y);
    return Py_BuildValue("");
}
```

```

static PyMethodDef PtExampleMethods[] = {
    {"print_point", print_point, METH_VARARGS, "output a point"},
    { NULL, NULL, 0, NULL}
};

static struct PyModuleDef ptexamplemodule = {
PyModuleDef_HEAD_INIT,
    "ptexample",
    /* Имя модуля */
    "A module that imports an API",
    /* Стока документации (может быть NULL) */
    -1,
    /* Размер состояния на каждый интерпретатор или -1 */
    PtExampleMethods
    /* Таблица методов */
};

/* Функция инициализации модуля */
PyMODINIT_FUNC
PyInit_ptexample(void) {
    PyObject *m;

    m = PyModule_Create(&ptexamplemodule);
    if (m == NULL)
        return NULL;
    /* Импортирует пример, загружает его функции API */
    if (!import_sample()) {
        return NULL;
    }
    return m;
}

```

При компилировании этого нового модуля вам даже не нужно беспокоиться о том, чтобы прилинковать библиотеки или код из другого модуля. Например, вы просто можете создать простой файл *setup.py*:

```

# setup.py
from distutils.core import setup, Extension

setup(name='ptexample',
      ext_modules=[
          Extension('ptexample',
                    ['ptexample.c'],
                    include_dirs = [],
                    # Может потребовать каталога, где лежит pysample.h
                    )
      ]
)

```

Если всё это зафурычит, вы обнаружите, что ваша новая функция расширения отлично работает с функциями С API, определёнными в другом модуле:

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p1
<capsule object "Point *" at 0x1004ea330>
>>> import ptexample
>>> ptexample.print_point(p1)
2.000000 3.000000
>>>
```

Обсуждение

Этот рецепт полагается на тот факт, что объекты капсул могут содержать указатель на всё, что угодно. В том случае определяющий модуль заполняет структуру указателей функций, создаёт капсулу, которая указывает на неё, а затем сохраняет капсулу в атрибуте уровня модуля (например, `sample._point_api`).

Другие модули могут быть запрограммированы «подобрать» этот атрибут при импортировании и извлечь указатель. На самом деле, Python предоставляет вспомогательную функцию `PyCapsule_Import()`, которая выполняет за вас все эти шаги. Вы просто передаёте ей имя атрибута (например, `sample._point_api`), и она найдёт капсулу и извлечёт указатель за один шаг.

Есть несколько приёмов программирования на С, которые используются для того, чтобы заставить экспортированные функции выглядеть обычными в других модулях. В файле `pysample.h` указатель `_point_api` использован, чтобы указать на таблицу методов, которая была инициализирована в экспортирующем модуле. Связанная функция `import_sample()` использована, чтобы выполнить требуемый импорт капсулы и инициализирует этот указатель. Функция должна быть вызвана перед использованием любых других функций. В обычном случае она будет вызвана во время инициализации модуля. Наконец, набор макросов для препроцессинга на С определены для прозрачной диспетчеризации функций API по таблице методов. Пользователь просто вызывает изначальные имена функций, но не знает о дополнительном перенаправлении через эти макросы.

Наконец, есть еще одна важная причина использовать приём линковки

модулей вместе — это проще и поддерживает модули в более чистом состоянии слабой связанности. Если вы не хотите использовать этот рецепт так, как показано, вы можете перекрестно слинковать модули, используя продвинутые возможности разделяемых библиотек и динамический загрузчик. Например, можно поместить обычные функции API в разделяемую библиотеку и убедиться, что все модули расширения слинкованы с этой библиотекой. По сути, этот рецепт вырезает всю магию и позволяет модулям линковаться друг к другу через обычный механизм импортирования Python и небольшое количество вызовов капсул. Для компилирования модулей вам нужно позаботиться только о заголовочных файлах, но не о кучерявых подробностях, касающихся разделяемых библиотек.

Дополнительную информацию о предоставлении С API модулям расширений можно найти в [документации Python](#).

15.6. Вызываем Python из С

Задача

Вы хотите безопасно выполнить вызываемый объект Python и вернуть результат обратно в С. Возможно, например, вы пишете код на С, в котором хотите использовать функцию Python в качестве коллбэка.

Решение

Вызов Python из С по большей части выполняется без каких-либо ухищрений, но включает и несколько тонких моментов. Следующий пример на С показывает, как можно сделать это безопасно:

```
#include <Python.h>

/* Выполняет func(x,y) в интерпретаторе Python. Аргументы
и возвращаемый результат функции должны быть числами
с плавающей точкой (Python floats). */

double call_func(PyObject *func, double x, double y) {
    PyObject *args;
    PyObject *kwargs;
    PyObject *result = 0;
    double retval;
```

```

/* Проверяем, что мы владеем GIL */
PyGILState_STATE state = PyGILState_Ensure();

/* Проверяем, что func — подходящий вызываемый объект */
if (!PyCallable_Check(func)) {
    fprintf(stderr,"call_func: expected a callable\n");
    goto fail;
}

/* Строим аргументы */
args = Py_BuildValue("(dd)", x, y);
kargs = NULL;

/* Вызываем функцию */
result = PyObject_Call(func, args, kargs);
Py_DECREF(args);
Py_XDECREF(kargs);

/* Проверяем на исключения Python (если они есть) */
if (PyErr_Occurred()) {
    PyErr_Print();
    goto fail;
}

/* Проверяем, что результат — объект float */
if (!PyFloat_Check(result)) {
    fprintf(stderr,"call_func: callable didn't return a float\n");
    goto fail;
}

/* Создаём возвращаемое значение */
retval = PyFloat_AsDouble(result);
Py_DECREF(result);

/* Восстанавливаем предыдущее состояние GIL и возвращаем значение */
PyGILState_Release(state);
return retval;

fail:
Py_XDECREF(result);
PyGILState_Release(state);
abort(); // Изменить на что-то более приемлемое
}

```

Чтобы использовать эту функцию, вы должны получить ссылку на существующий вызываемый объект Python, чтобы передать её ей. Есть много способов это сделать — например, передать вызываемый объект в модуль расширения или просто написать код на C, чтобы извлечь символ из существующего модуля.

Вот простой пример, который демонстрирует вызов функции из встроенного интерпретатора Python:

```
#include <Python.h>

/* Определение call_func() такое же, как и выше */
...

/* Загрузить символ из модуля */
PyObject *import_name(const char *modname, const char *symbol) {
    PyObject *u_name, *module;
    u_name = PyUnicode_FromString(modname);
    module = PyImport_Import(u_name);
    Py_DECREF(u_name);
    return PyObject_GetAttrString(module, symbol);
}

/* Простой пример встраивания */
int main() {
    PyObject *pow_func;
    double x;

    Py_Initialize();
    /* Получить ссылку на функцию math.pow */
    pow_func = import_name("math", "pow");

    /* Вызвать её, используя наш код call_func() */
    for (x = 0.0; x < 10.0; x += 0.1) {
        printf("%0.2f %0.2f\n", x, call_func(pow_func,x,2.0));
    }
    /* Готово */
    Py_DECREF(pow_func);
    Py_Finalize();
    return 0;
}
```

Чтобы скомпилировать этот последний пример, вы должны скомпилировать код С и слинковать его с интерпретатором Python. Вот make-файл, который показывает, как вы можете это сделать (это может потребовать некоторых танцев с бубном, чтобы всё запустилось на вашем компьютере):

```
all:::
    cc -g embed.c -I/usr/local/include/python3.3m \
        -L/usr/local/lib/python3.3/config-3.3m -lpython3.3m
```

Компиляция и запуск получившегося исполняемого файла создаст похожий на этот вывод:

```
0.00 0.00
0.10 0.01
0.20 0.04
0.30 0.09
0.40 0.16
...
```

А вот немного отличающийся пример, который демонстрирует функцию расширения, которая принимает вызываемый объект и аргументы, а затем передаёт их в функцию *call_func()* для целей тестирования:

```
/* Функция расширения для проверки коллбэка C-Python */
PyObject *py_call_func(PyObject *self, PyObject *args) {
    PyObject *func;
    double x, y, result;
    if (!PyArg_ParseTuple(args, "Odd", &func, &x, &y)) {
        return NULL;
    }
    result = call_func(func, x, y);
    return Py_BuildValue("d", result);
}
```

Эту функцию расширения вы можете протестировать так:

```
>>> import sample
>>> def add(x,y):
...     return x+y
...
>>> sample.call_func(add,3,4)
7.0
>>>
```

Обсуждение

Если вы вызываете Python из С, самое важное — помнить о том, что С в общем случае должен всегда быть «главным». На С лежит ответственность за создание аргументов, вызов функций Python, проверку на исключения, проверку типов, извлечение возвращённых значений и т.д.

В качестве первого шага критически важно иметь объект Python, представляющий вызываемый объект, который вы будете вызывать. Это может быть функция, класс, метод, встроенный метод или всё, что реализует операцию *__call__()*. Чтобы убедиться, что объект можно вызвать, используйте *PyCallable_Check()*, как показано в этом фрагменте кода:

```

double call_func(PyObject *func, double x, double y) {
    ...
    /* Проверяем, что func — правильный вызываемый объект */
    if (!PyCallable_Check(func)) {
        fprintf(stderr,"call_func: expected a callable\n");
        goto fail;
    }
    ...

```

Отметим, что обработка ошибок в коде на С требует внимательного изучения. В качестве общего правила: вы не можете просто возбудить исключение Python. Вместо этого ошибки должны быть обработаны способом, который имеет смысл для вашего кода на С. В решении мы используем переход *goto*, чтобы передать управление блоку обработки ошибок, который вызывает *abort()*. Это вызывает завершение всей программы, но в настоящем коде вы, вероятно, захотите делать это более аккуратно (например, возвращать код статуса). Держите в голове, что С тут главный, и в нём нет ничего похожего на возбуждение исключения. Обработку ошибок нужно реализовывать в конкретной программе отдельно.

Вызов функции относительно бесхитростен — просто используйте *PyObject_Call()*, предоставив ей вызываемый объект, кортеж аргументов и необязательный словарь именованных аргументов. Чтобы создать кортеж аргументов или словарь, вы можете применить *Py_BuildValue()*.

```

double call_func(PyObject *func, double x, double y) {
    PyObject *args;
    PyObject *kwargs;
    ...

    /* Строим аргументы */
    args = Py_BuildValue("(dd)", x, y);
    kwargs = NULL;

    /* Вызываем функцию */
    result = PyObject_Call(func, args, kwargs);
    Py_DECREF(args);
    Py_XDECREF(kwargs);
    ...

```

Если у вас нет именованных аргументов, вы можете передать `NULL`. После выполнения вызова функции, вам нужно убедиться, что вы подчистили аргументы, используя *Py_DECREF()* или *Py_XDECREF()*. Последняя функция безопасно позволяет указателю `NULL` быть переданным (что игнорируется), и

поэтому мы используем её для подчистки необязательных ключевых аргументов.

После вызова функции Python вы должны проверить, не возникли ли исключения. Для этого вам пригодится функция *PyErr_Occurred()*. Но определить, что сделать в ответ на исключение, достаточно трудно. Поскольку вы работаете из C, у вас просто нет всей системы исключений, которая есть в Python. Так что вы можете установить код статуса ошибки, логировать ошибку или выполнить какую-то другую имеющую смысл обработку. В решении этого рецепта при отсутствии простой альтернативы вызывается *abort()* (чёткие C-разработчики оценят крутое падение программы):

```
...
/* Проверка на исключения Python (если они есть) */
if (PyErr_Occurred()) {
    PyErr_Print();
    goto fail;
}
...
fail:
    PyGILState_Release(state);
    abort();
```

Получение информации из возвращённого функцией Python значения в типичном случае потребует некоторой проверки типов и извлечения значения. Чтобы сделать это, вы можете использовать функции из [слоя конкретных объектов Python](#). В решении код проверяет тип числа с плавающей точкой и извлекает его значение с помощью *PyFloat_Check()* и *PyFloat_AsDouble()*.

Последняя сложная часть вызова Python из C касается управления глобальной блокировкой интерпретатора (GIL) Python. Когда бы вы ни обращались к Python из C, вам нужно убедиться, что GIL правильно получается и освобождается. В противном случае вы столкнётесь с ситуацией, когда интерпретатор портит данные или падает. Вызовы *PyGILState_Ensure()* и *PyGILState_Release()* позволяют убедиться, что всё сделано правильно:

```
double call_func(PyObject *func, double x, double y) {
    ...
    double retval;
```

```
/* Убеждаемся, что владеем GIL */
PyGILState_STATE state = PyGILState_Ensure();
...
/* Код, который использует функции Python C API */
...
/* Восстанавливаем предыдущее состояние GIL и возвращаем значение */
PyGILState_Release(state);
return retval;

fail:
PyGILState_Release(state);
abort();
}
```

До завершения выполнения `PyGILState_Ensure()` всегда гарантирует, что вызывающий поток имеет эксклюзивный доступ к интерпретатору Python. Это останется так, даже если вызывающий код на С запустит другой поток, который неизвестен интерпретатору. В этой точке код на С свободен использовать любые функции Python C API, какие пожелает. После успешного завершения `PyGILState_Release()` используется для восстановления изначального состояния интерпретатора.

Критически важно отметить, что за каждым вызовом `PyGILState_Ensure()` должен следовать соответствующий вызов `PyGILState_Release()` — даже если возникли ошибки. В решении инструкция `goto` может выглядеть, как пример ужасного проектирования, но мы используем её для передачи управления обычному блоку вызода, который выполняет этот требуемый шаг. Думайте о коде после `fail:`, как об аналоге блока `finally:` в Python.

Если вы пишете ваш код на С, используя все эти соглашения, включая управление GIL, проверку на исключения и тщательную проверку ошибок, то вы обнаружите, что вы можете вполне надёжно вызывать интерпретатор Python из С — даже в очень сложных программах, которые используют продвинутые приёмы программирования (такие, как многопоточность).

15.7. Освобождение GIL в расширениях на С

Задача

У вас есть код расширения на С, который вы хотите конкурентно выполнять с

другими потоками в интерпретаторе Python. Чтобы сделать это, вам нужно освобождать и снова получать глобальную блокировку интерпретатора (GIL).

Решение

В коде расширения на С можно освобождать и снова приобретать GIL путём помещения в код следующего макроса:

```
#include "Python.h"
...
PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Многопоточный код на С. Нельзя использовать функции Python API
    ...
    Py_END_ALLOW_THREADS
    ...
    return result;
}
```

Обсуждение

GIL может быть безопасно освобождён, только если вы можете гарантировать, что в коде на С не будут выполняться функции Python C API. Типичные примеры освобождения GIL — это код, выполняющий сложные вычисления над массивами С (например, в расширениях типа *pintru*), или код, где выполняются блокирующие операции ввода-вывода (например, чтение или запись в файловый дескриптор).

Пока GIL освобождён, другие потоки Python могут выполняться в интерпретаторе. Макрос *Py-END_ALLOW_THREADS* блокирует выполнение, пока вызывающие потоки не получат в интерпретаторе GIL обратно.

15.8. Смешивание потоков из С и Python

Задача

У вас есть программа, которая представляет собой смесь С, Python и потоков, причём какие-то потоки создаются из С, за пределами контроля интерпретатора Python. Более того, некоторые потоки используют функции

Python C API.

Решение

Если вы собираетесь замиксовывать C, Python и потоки, то вам нужно убедиться, что вы правильно инициализируете глобальную блокировку интерпретатора Python (GIL) и управляете ей. Чтобы сделать это, включите следующий код куда-то в вашу программу на C — и убедитесь, что вызываете его перед созданием каких-либо потоков:

```
#include <Python.h>
...
if (!PyEval_ThreadsInitialized()) {
    PyEval_InitThreads();
}
...
...
```

Для любого кода на C, который использует объекты Python или Python C API, убедитесь для начала, что вы правильно приобретаете и освобождаете GIL. Это делается с помощью *PyGILState_Ensure()* и *PyGILState_Release()*, как показано тут:

```
...
/* Убедимся, что владеем GIL */
PyGILState_STATE state = PyGILState_Ensure();

/* Используем функции в интерпретаторе */
...
/* Восстанавливаем предыдущее состояние GIL и возвращаем значение */
PyGILState_Release(state);
...
```

За каждым вызовом *PyGILState_Ensure()* должен следовать соответствующий вызов *PyGILState_Release()*.

Обсуждение

В продвинутых приложениях на базе C и Python множество одновременных событий — не такое уж редкое дело. И часто это подразумевает смесь кода на C, Python, потоков C и потоков Python. Пока вы тщательно проверяете правильность инициализации интерпретатора и правильность вызовов для управления GIL из кода на C, всё должно работать.

Обратите внимание, что вызов `PyGILState_Ensure()` не мгновенно завладевает интерпретатором или прерывает его. Если другой код находится в состоянии исполнения, эта функция заблокируется до тех пор, пока этот код не решит освободить GIL. Внутри интерпретатор выполняет периодическое переключение потоков, так что даже если другой поток выполняется, вызывающий рано или поздно запустится (хотя ему, возможно, придётся сначала немного подождать).

15.9. Оборачивание кода на С в Swig

Задача

У вас есть существующий код на С, к которому вы хотели бы получить доступ, как к модулю расширения на С. Мы можем сделать это с помощью [генератора обёрток Swig](#).

Решение

Swig работает путём парсинга заголовочных файлов С и автоматического создания кода расширения. Чтобы использовать его, вам сначала нужно иметь заголовочный файл С. Например, это заголовочный файл нашего кода-примера:

```
/* sample.h */

#include <math.h>
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

Если у вас есть заголовочные файлы, следующим шагом будет написание файла с «интерфейсом» Swig. Эти файлы имеют расширение `.i` и могут выглядеть как-то так:

```

// sample.i - Swig interface
%module sample
%{
#include "sample.h"
%}

/* Кастомизации */
%extend Point {
    /* Конструктор объектов Point */
    Point(double x, double y) {
        Point *p = (Point *) malloc(sizeof(Point));
        p->x = x;
        p->y = y;
        return p;
    };
};

/* Отобразить int *remainder как выходной аргумент */
%include typemaps.i
%apply int *OUTPUT { int * remainder };

/* Отобразить шаблон аргументов (double *a, int n) на массивы */
%typemap(in) (double *a, int n)(Py_buffer view) {
    view.obj = NULL;
    if (!PyObject_GetBuffer($input, &view, PyBUF_ANY_CONTIGUOUS | PyBUF_I
        SWIG_fail;
    }
    if (strcmp(view.format, "d") != 0) {
        PyErr_SetString(PyExc_TypeError, "Expected an array of doubles")
        SWIG_fail;
    }
    $1 = (double *) view.buf;
    $2 = view.len / sizeof(double);
}

%typemap(freearg) (double *a, int n) {
    if (view$argnum.obj) {
        PyBuffer_Release(&view$argnum);
    }
}

/* Объявления С для включения в модуль расширения */
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

```

```
extern double distance(Point *p1, Point *p2);
```

Когда вы написали файл интерфейса, Swig вызывается в качестве инструмента командной строки:

```
bash % swig -python -py3 sample.i  
bash %
```

На выходе команды *swig* получается два файла: *sample_wrap.c* и *sample.py*. Последний файл — это то, что импортирует пользователь. *sample_wrap.c* — это код на С, который должен быть скомпилирован в поддерживающий модуль под названием *_sample*. Это делается с использованием тех же приёмов, что и для обычных модулей расширения. Например, вы создаете файл *setup.py* таким образом:

```
# setup.py  
from distutils.core import setup, Extension  
  
setup(name='sample',  
      py_modules=[ 'sample.py' ],  
      ext_modules=[  
          Extension('_sample',  
                    [ 'sample_wrap.c' ],  
                    include_dirs = [],  
                    define_macros = [],  
                    undef_macros = [],  
                    library_dirs = [],  
                    libraries = [ 'sample' ]  
      )  
]
```

Чтобы скомпилировать и всё проверить, запустите *setup.py* в *python3*:

```
bash % python3 setup.py build_ext --inplace  
running build_ext  
building '_sample' extension  
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes  
-I/usr/local/include/python3.3m -c sample_wrap.c  
-o build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o  
sample_wrap.c: In function 'SWIG_InitializeModule':  
sample_wrap.c:3589: warning: statement with no effect  
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/  
build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o -o _sample.so -lsample  
bash %
```

Если всё получится, вы обнаружите, что можете использовать получившийся модуль расширения С самым прямолинейным образом. Например:

```
>>> import sample
>>> sample.gcd(42,8)
2
>>> sample.divide(42,8)
[5, 2]
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
2.8284271247461903
>>> p1.x
2.0
>>> p1.y
3.0
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>>
```

Обсуждение

Swig — это один из самых старых инструментов для создания модулей расширения, история которого восходит к Python 1.4. Однако текущие версии поддерживают Python 3. В основном пользователями Swig становятся программисты, у которых есть обширная кодовая база на C, к которой они хотят получить доступ, используя Python в качестве высокоуровневого языка управления. Например, пользователь может иметь код на C, содержащий тысячи функций и разнообразные структуры данных, к которым он хотел бы получить доступ из Python. Swig может автоматизировать большую часть процесса генерации обёрток.

Все интерфейсы Swig склонны начинаться с короткого введения:

```
%module sample
%{
#include "sample.h"
%}
```

Это просто объявляет имя модуля расширения и определяет заголовочные файлы C, которые должны быть включены, чтобы всё скомпилировалось

(код, заключённый между `%{ %}`, напрямую вставляется в выводимый код, так что сюда вам нужно поместить все включённые файлы и другие определения, необходимые для компиляции).

В конце интерфейса Swig находится список объявлений C, которые нужно включить в расширение. Он часто просто копируется из заголовочных файлов. В нашем примере мы просто вставляем заголовочный файл напрямую:

```
%module sample
%{
#include "sample.h"
%}
...
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

Важно подчеркнуть, что эти объявления сообщают Swig о том, что вы хотите включить в модуль Python. Очень часто этот список объявлений редактируют, чтобы внести необходимые изменения. Например, если вы не хотите включать некоторые объявления, вы можете удалить их из списка объявлений.

Самый сложный момент работы со Swig — это разнообразные кастомизации, которые он может применить к коду на C. Это огромная тема, которую мы не можем тут детально разбирать, но несколько таких кастомизаций в рецепте показаны.

Первая использует директиву `%extend` и позволяет методам прикрепляться к существующим структурам и определениям классов. В примере это используется для добавления метода-конструктора к структуре `Point`. Эта кастомизация делает возможным использовать структуру таким образом:

```
>>> p1 = sample.Point(2,3)
>>>
```

Если это опустить, объекты *Point* придётся создавать намного более неуклюжим способом:

```
>>> # Использование, если %extend Point опущено
>>> p1 = sample.Point()
>>> p1.x = 2.0
>>> p1.y = 3
```

Вторая кастомизация подразумевает включение библиотеки *turmaps.i* и директивы *%apply*, которая сообщает Swig, что с аргументной сигнатурой *int *remainder* нужно обращаться, как с выходным (*output*) значением. На самом деле, это правило сопоставления с образцом (*pattern matching*). Во всех следующих объявлениях при встрече с *int *remainder* с ним будут обращаться как с выводом (*output*). Эта кастомизация заставляет функцию *divide()* возвращать два значения:

```
>>> sample.divide(42, 8)
[5, 2]
>>>
```

Последняя кастомизация, использующая директиву *%туретар*, вероятно, самая продвинутая из показанных нами. Отображение типа (*туретар*) — это правило, которое применяется к конкретным паттернам аргументов ввода (*input*). В этом рецепте *туретар* написан для сопоставления с паттерном аргументов (*double *a, int n*). Внутри *туретар* представляет собой фрагмент кода на C, который указывает Swig, как преобразовать объект Python в ассоциированные аргументы C. Код этого рецепта написан с использованием буферного протокола Python, чтобы попытаться сопоставить любые входные аргументы, которые выглядят, как массивы чисел с двойной точностью (например, массивы NumPy, массивы модуля array и т.п.) См. [рецепт 15.3](#).

Внутри кода *туретар* такие подстановки, как *\$1* и *\$2*, ссылаются на переменные, которые содержат преобразованные в паттерны *туретар* значения аргументов C (например, *\$1* отображается на *double *a*, а *\$2* отображается на *int n*). *\$input* ссылается на аргумент *PyObject **, который был предоставлен как аргумент ввода (*input argument*). *\$argnum* — это номер аргумента.

Написать и понять *туретар*'ы — одна из самых трудных, часто даже гибельно трудных задач для программистов, использующих Swig. Мало того, что код весьма загадочен — вам нужно также понимать замысловатые детали C API

Python и того, как Swig с ним взаимодействует. В документации Swig приведено много примеров и подробная информация.

Несмотря на это, если у вас много кода на C, который нужно использовать как модуль расширения, Swig может стать очень мощным инструментом. Главное — держать в уме, что Swig, по сути, является компилятором, обрабатывающим объявления C, но с мощным механизмом сопоставления с образцом (pattern matching) и компонентом для кастомизации, который позволяет вам менять способ, которым обрабатываются конкретные объявления и типы. Больше сведений вы найдёте на [сайте Swig](#). Есть там и [документация по работе с Python](#).

15.10. Оборачивание существующего кода на С в Cython

Задача

Вы хотите использовать [Cython](#), чтобы заставить модуль расширения Python обернуться вокруг существующей библиотеки на C.

Решение

Создание модуля расширения с помощью Cython выглядит похоже на создание самописного расширения, что подразумевает создание коллекции функций-обёрток. Однако, в отличие от предыдущих рецептов, вы не будете делать это на C — код будет выглядеть намного более похожим на код Python.

В качестве предварительного условия предположим, что код примера, показанного во введении в эту главу, скомпилирован в библиотеку на C под названием *libsample*. Начнём с создания файла с именем *csample.pxd*, который выглядит так:

```
# csample.pxd
#
# Объявления "внешних" функций и структур С

cdef extern from "sample.h":
    int gcd(int, int)
    bint in_mandel(double, double, int)
```

```

int divide(int, int, int *)
double avg(double *, int) nogil

ctypedef struct Point:
    double x
    double y

double distance(Point *, Point *)

```

Этот файл служит в Cython тем же целям, что и заголовочный файл в C.

Первоначальное объявление `cdef extern from "sample.h"` объявляет требуемый заголовочный файл C. Следующие объявления взяты из этого заголовка. Имя этого файла — `csample.pxd`, а не `sample.pxd`. Это важно.

На следующем шаге мы создаём файл с именем `sample.pyx`. Этот файл определяет обёртки, которые соединяют интерпретатор Python с лежащим в основе кодом на C, определённом в файле `csample.pxd`:

```

# sample.pyx

# Импортируем низкоуровневые объявления C
cimport csample

# Импортируем некую функциональность из Python и C stdlib
from cpython.pycapsule cimport *
from libc.stdlib cimport malloc, free

# Обёртки
def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x, y)

def in_mandel(x, y, unsigned int n):
    return csample.in_mandel(x, y, n)

def divide(x, y):
    cdef int rem
    quot = csample.divide(x, y, &rem)
    return quot, rem

def avg(double[:] a):
    cdef:
        int sz
        double result

    sz = a.size
    with nogil:
        result = csample.avg(<double *> &a[0], sz)
    return result

```

```

# Деструктор для зачистки объектов Point
cdef del_Point(object obj):
    pt = <csample.Point *> PyCapsule_GetPointer(obj,"Point")
    free(<void *> pt)

# Создание объекта Point и возвращение в форме капсулы
def Point(double x,double y):
    cdef csample.Point *p
    p = <csample.Point *> malloc(sizeof(csample.Point))
    if p == NULL:
        raise MemoryError("No memory to make a Point")
    p.x = x
    p.y = y
    return PyCapsule_New(<void *>p,"Point",<PyCapsule_Destructor>del_Poi

def distance(p1, p2):
    pt1 = <csample.Point *> PyCapsule_GetPointer(p1,"Point")
    pt2 = <csample.Point *> PyCapsule_GetPointer(p2,"Point")
    return csample.distance(pt1,pt2)

```

Различные детали этого файла мы разберём в разделе «Обсуждение» этого рецепта.

Наконец, чтобы собрать модуль расширения, создадим файл setup.py, который выглядит так:

```

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',
              ['sample.pyx'],
              libraries=['sample'],
              library_dirs=['.'])]
setup(
    name = 'Sample extension module',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)

```

Чтобы собрать получившийся модуль, напечатайте нижеследующее:

```

bash % python3 setup.py build_ext --inplace
running build_ext
cythoning sample.pyx to sample.c

```

```
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
-I/usr/local/include/python3.3m -c sample.c
-o build/temp.macosx-10.6-x86_64-3.3/sample.o
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3,
-L. -lsample -o sample.so
bash %
```

Если это заработает, вы должны получить модуль расширения `sample.so`, который может быть использован так, как показано в примере:

```
>>> import sample
>>> sample.gcd(42,10)
2
>>> sample.in_mandel(1,1,400)
False
>>> sample.in_mandel(0,0,400)
True
>>> sample.divide(42,10)
(4, 2)
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1005d1e70>
>>> p2
<capsule object "Point" at 0x1005d1ea0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

Обсуждение

Этот рецепт объединяет многие продвинутые возможности, обсуждавшиеся в предыдущих рецептах — манипулирование массивами, обрачивание непрозрачных указателей, освобождение GIL. Каждая из этих частей будет рассмотрена далее, но для начала вы можете ещё раз прочитать предудыщие рецепты.

На высоком уровне использование Cython построено по модели С. Файлы `.pxd` просто содержат определения С (похожим на файлы `.h` образом), а файлы `.pyx` содержат реализацию (похожим на файлы `.c` образом).

Инструкция `cimport` используется Cython для импортирования определений из файла `.pxd`. Это отличается от использования обычной инструкции `import`, которая загружает обычный модуль Python.

Хотя файлы `.pxd` содержат определения, они не используются для цели автоматического создания кода расширения. Вам всё равно придётся писать простые функции-обёртки. Например, хотя файл `csample.pxd` объявляет `int gcd(int, int)` как функцию, вам всё равно придётся написать небольшую обёртку для неё в `sample.pxd`. Например:

```
cimport csample

def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x,y)
```

Для простых функций особо напрягаться не придётся. Cython сгенерирует обрабатывающий код, который правильно преобразует аргументы и возвращаемое значение. Типы данных C, прикреплённые к аргументам, являются необязательными. Однако если вы их включите, то получите дополнительную бесплатную проверку ошибок. Например, если кто-то вызовет эту функцию с отрицательными значениями, будет сгенерировано исключение:

```
>>> sample.gcd(-10,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.pyx", line 7, in sample.gcd (sample.c:1284)
    def gcd(unsigned int x,unsigned int y):
OverflowError: can't convert negative value to unsigned int
>>>
```

Если вы хотите добавить дополнительную проверку в обёртку, просто используйте дополнительный обрабатывающий код. Например:

```
def gcd(unsigned int x, unsigned int y):
    if x <= 0:
        raise ValueError("x must be > 0")
    if y <= 0:
        raise ValueError("y must be > 0")
    return csample.gcd(x,y)
```

Объявление `in_mandel()` в файле `csample.pxd` — интересный и тонкий момент. В этом файле функция определена как возвращающая `bint` вместо `int`. Это

заставляет функцию создавать правильное булево значение из результата вместо простого целого числа. Так что возвращаемое значение 0 отображается в `False`, а 1 — в `True`.

Внутри обёрток Cython у вас есть необязательная возможность декларировать типы данных C в дополнение к использованию обычных объектов Python. Обёртка над `divide()` показывает пример этого, а также пример работы с аргументом-указателем.

```
def divide(x,y):
    cdef int rem
    quot = csample.divide(x,y,&rem)
    return quot, rem
```

Здесь переменная `rem` явно объявлена как переменная C `int`. При передаче в функцию `divide()`, `&rem` создает указатель на неё — так же, как в C.

Код функции `avg()` иллюстрирует некоторые более продвинутые возможности Cython. Во-первых, объявление `def avg(double[:] a)` объявляет `avg()` как принимающую одномерное представление памяти (`memoryview`) значений `double`. Интересно, что получившаяся функция будет принимать любой совместимый объект массива, включая созданные библиотеками типа `numpy`. Например:

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> import numpy
>>> b = numpy.array([1., 2., 3.])
>>> import sample
>>> sample.avg(a)
2.0
>>> sample.avg(b)
2.0
>>>
```

В обёртке `a.size` и `&a[0]` ссылаются, соответственно, на количество элементов массива и лежащий в основе указатель. С помощью синтаксиса `\&a[0]` вы при необходимости приводите указатели к другому типу. Это нужно, чтобы убедиться, что C `avg()` принимает указатель корректного типа. Отсылаем вас к следующему рецепту, где рассматривается более продвинутый пример применения представлений памяти (`memoryviews`) Cython.

В дополнение к работе с обобщенными массивами, пример `avg()` также

показывает, как работать с глобальной блокировкой интерпретатора. Инструкция *with nogil*: объявляет блок кода, как выполняющийся без GIL. Внутри этого блока нельзя работать ни с какими обычными объектами Python, можно использовать только объекты и функции, определённые как *cdef*. В дополнение к этому, внешние функции должны явно определять, что они могут выполняться без GIL. Так, в файле *csample.rxd* мы определяем *avg()* как *double avg(double *, int) nogil*.

Работа со структурой *Point* — это отдельный вызов способностям программиста. Как показано выше, этот рецепт обращается с объектами *Point* как непрозрачными указателями, используя объекты капсул, как описано в **рецепте 15.4**. Однако чтобы сделать это, лежащий в основе код Cython будет немного посложнее. Во-первых, ниже приведённые операции импортирования используются для переноса определений функций из библиотеки C и Python C API:

```
from cpython.pycapsule cimport *
from libc.stdlib cimport malloc, free
```

Функция *del_Point()* и *Point()* используют эту функциональность для создания объекта капсулы, который обрамляется вокруг указателя *Point **. Объявление *cdef del_Point()* объявляет *del_Point()* как функцию, которая доступна только из Cython, но не из Python. Также эта функция не будет видима извне — вместо этого она используется как функция обратного вызова (коллбэк) для очистки памяти, выделенной капсулой. Вызовы таких функций, как *PyCapsule_New()* и *PyCapsule_GetPointer()*, выполняются напрямую из Python C API и используются так же.

Функция *distance()* написана, чтобы извлекать указатели из объектов капсул, созданных *Point()*. Стоит отметить, что вам не нужно волноваться об обработке исключений. В случае получения «плохого» объекта *PyCapsule_GetPointer()* возбуждает исключение, но Cython уже знает, каким образом на него смотреть, и распространяет его за пределы функции *distance()*, если оно возникает.

Недостаток работы со структурами *Point* заключается в том, что они в этой реализации будут полностью непрозрачными. Вы не сможете подсмотреть, как они устроены, или получить доступ к их атрибутам. Есть альтернативный подход к обрамлению, который состоит в определении типа расширения, как показано в этом коде:

```

# sample.pyx

cimport csample
from libc.stdlib cimport malloc, free
...

cdef class Point:
    cdef csample.Point *_c_point
    def __cinit__(self, double x, double y):
        self._c_point = <csample.Point *> malloc(sizeof(csample.Point))
        self._c_point.x = x
        self._c_point.y = y

    def __dealloc__(self):
        free(self._c_point)

    property x:
        def __get__(self):
            return self._c_point.x
        def __set__(self, value):
            self._c_point.x = value

    property y:
        def __get__(self):
            return self._c_point.y
        def __set__(self, value):
            self._c_point.y = value

    def distance(Point p1, Point p2):
        return csample.distance(p1._c_point, p2._c_point)

```

Здесь `cdef class Point` объявляет `Point` как тип расширения. Переменная класса `cdef csample.Point *_c_point` объявляет переменную экземпляра, которая содержит указатель на лежащую в основе `Point` структуру на С. Методы `__cinit__()` и `__cinit__()` создают и разрушают лежащую в основе структуру С с помощью вызовов `malloc()` и `free()`. Объявления `property x` и `property y` создают код, который получает и устанавливает атрибуты лежащей в основе структуры. Обёртка для `distance()` также была удобно модифицирована, чтобы принимать экземпляры типа расширения `Point` как аргументы, но передавать лежащий в основе указатель функции на С.

Внеся это изменение, вы обнаружите, что код для манипулирования объектами `Point` стал более естественным:

```

>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)

```

```
>>> p1
<sample.Point object at 0x100447288>
>>> p2
<sample.Point object at 0x1004472a0>
>>> p1.x
2.0
>>> p1.y
3.0
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

Этот рецепт демонстрирует многие ключевые возможности Cython, которые вы можете экстраполировать на более сложные типы обрачивания (wrapping). Однако вам точно придётся почитать [официальную документацию](#).

Следующие несколько примеров также демонстрируют дополнительные возможности Cython.

15.11. Использование Cython для высокопроизводительных операций над массивами

Задача

Вы хотите написать функции для высокопроизводительной обработки массивов, чтобы работать с массивами из библиотек типа NumPy. Вы слышали, что инструменты типа Cython облегчают эту задачу, но не знаете, с чего начать.

Решение

В качестве примера рассмотрим следующий код, показывающий функцию Cython для вырезания значений из простого одномерного массива чисел с двойной точностью:

```
# sample.pyx (Cython)

cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
```

```

cpdef clip(double[:] a, double min, double max, double[:] out):
    """
    Вырезает значения в a так, чтобы они были между min и max. Результат в out
    """

    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    for i in range(a.shape[0]):
        if a[i] < min:
            out[i] = min
        elif a[i] > max:
            out[i] = max
        else:
            out[i] = a[i]

```

Чтобы скомпилировать и собрать расширение, вам потребуется такой файл `setup.py` (для сборки используйте `python3 setup.py build_ext --inplace`):

```

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',
              ['sample.pyx'])
]

setup(
    name = 'Sample app',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)

```

Вы обнаружите, что получившаяся функция обрезает массивы и работает с многими различными типами объектов массивов. Например:

```

# Пример с модулем array
>>> import sample
>>> import array
>>> a = array.array('d',[1,-3,4,7,2,0])
>>> a
array('d', [1.0, -3.0, 4.0, 7.0, 2.0, 0.0])
>>> sample.clip(a,1,4,a)
>>> a
array('d', [1.0, 1.0, 4.0, 4.0, 2.0, 1.0])

>>> # Пример с numpy

```

```
>>> import numpy
>>> b = numpy.random.uniform(-10,10,size=1000000)
>>> b
array([-9.55546017, 7.45599334, 0.69248932, ..., 0.69583148,
       -3.86290931, 2.37266888])
>>> c = numpy.zeros_like(b)
>>> c
array([ 0., 0., 0., ..., 0., 0., 0.])
>>> sample.clip(b,-5,5,c)
>>> c
array([-5.          , 5.          , 0.69248932, ..., 0.69583148,
       -3.86290931, 2.37266888])
>>> min(c)
-5.0
>>> max(c)
5.0
>>>
```

Вы также обнаружите что получившийся код быстро работает. Следующий пример выводит нашу реализацию на бой с функцией *clip()* из *numpy*:

```
>>> timeit('numpy.clip(b,-5,5,c)', 'from __main__ import b,c,numpy', number=8.093049556000551
>>> timeit('sample.clip(b,-5,5,c)', 'from __main__ import b,c,sample',
...           number=1000)
3.760528204000366
>>>
```

Как вы можете видеть, она немного быстрее — интересный результат, если принять во внимание, что ядро версии на базе NumPy написано на C.

Обсуждение

Этот рецепт использует типизированные представления памяти Cython (typed memoryviews), которые значительно упрощают код для работы с массивами. Объявление *cpdef clip()* объявляет *clip()* сразу и как функцию уровня C, и как функцию уровня Python. В Cython это полезно, поскольку это означает, что вызов функции более эффективно вызывается другими функциями Cython (например, если вы хотите вызвать *clip()* из другой функции Cython).

Типизированные параметры *double[:] a* и *double[:] out* объявляют эти параметры как одномерные массивы чисел с двойной точностью. В качестве входных данных они будут осуществлять доступ к любому объекту массива, который правильно реализует интерфейс представления памяти

(`memoryview`), как описано в [PEP 3118](#). Это включает массивы из NumPy и из встроенной библиотеки `array`.

При написании кода, который выводит результат, который также является массивом, вы должны следовать показанным соглашениям о выходном (`output`) параметре. Это возлагает ответственность на создание выходного массива называющего и освобождает код от необходимости слишком много знать о конкретных деталях того, какими массивами он манипулирует (он просто предполагает, что массивы уже подходят, и нужно просто выполнить несколько базовых проверок, таких как проверка совместимости размера). В библиотеках типа NumPy относительно легко создавать выходные массивы с помощью функций типа `numpy.zeros()` или `numpy.zeros_like()`. В качестве альтернативы, чтобы создать неинициализированные массивы, вы можете использовать `numpy.empty()` или `numpy.empty_like()`. Это будет немного быстрее, если вы собираетесь перезаписать содержимое массива результатом.

В реализации вашей функции вы просто пишете прямолинейный код обработки массивов, используя индексирование и поиск в массиве (например, `a[i]`, `out[i]` и так далее). Cython предпримет шаги, чтобы убедиться в том, что это создает эффективный код.

Два декоратора, которые предшествуют определению `clip()`, являются необязательными оптимизациями производительности.

`@cython.boundscheck(False)` убирает все проверки границ массива и может быть использован, если вы знаете, что индексирование не выйдет за пределы диапазона. `@cython.wraparound(False)` убирает обработку отрицательных индексов массива как обёртку для индексирования с конца (как у списков Python). Включение этих декораторов может заставить код выполняться заметно быстрее (мы получили выигрыш в 2,5 раза при тестировании этого примера).

При работе с массивами внимательное изучение и эксперименты с лежащим в основе алгоритмом может привести к заметным выигрышам в скорости. Например, рассмотрите этот вариант функции `clip()`, который использует условные выражения:

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if min > max:
```

```
    raise ValueError("min must be <= max")
if a.shape[0] != out.shape[0]:
    raise ValueError("input and output arrays must be the same size")
for i in range(a.shape[0]):
    out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

При тестировании эта версия кода отработала на 50% быстрее (2,44 с против 3,76 с показанного ранее теста *timeit()*).

В этой точке вы можете задуматься, как этот код выступил бы против написанной вручную версии на С. Предположим, например, что вы пишете следующую функцию на С и вручную создаете расширение, используя приёмы из приведённых ранее рецептов:

```
void clip(double *a, int n, double min, double max, double *out) {
    double x;
    for (; n >= 0; n--, a++, out++) {
        x = *a;
        *out = x > max ? max : (x < min ? min : x);
    }
}
```

Код расширения для неё тут не показан, но после нескольких проверок мы обнаружили, что самостоятельно написанное с нуля расширение на С работает на 10% медленнее, чем версия, которую создал Cython. Короче говоря, код работает быстрее, чем вы могли бы предположить.

Есть несколько расширений, которые можно добавить в код решения. Для некоторых типов операций над массивами может иметь смысл освобождать GIL, чтобы несколько потоков могли работать параллельно. Чтобы сделать это, измените код, включив в него несколько инструкций *with nogil*:

Например:

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    with nogil:
        for i in range(a.shape[0]):
            out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

Если вы хотите написать версию кода, которая оперирует с двумерными массивами, то вот как он мог бы выглядеть:

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip2d(double[:, :] a, double min, double max, double[:, :] out):
    if min > max:
        raise ValueError("min must be <= max")
    for n in range(a.ndim):
        if a.shape[n] != out.shape[n]:
            raise TypeError("a and out have different shapes")
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            if a[i,j] < min:
                out[i,j] = min
            elif a[i,j] > max:
                out[i,j] = max
            else:
                out[i,j] = a[i,j]
```

Надеемся, что от читателя не ускользнуло, что весь код в этом рецепте не привязан к какой-либо конкретной библиотеке для работы с массивами (например, NumPy). Это даёт коду большую гибкость. Однако также стоит заметить, что работа с массивами может быть намного более сложной, если в игру вступают многомерность, размер шага, сдвиги и другие факторы. Обсуждение этих тем лежит за пределами этого рецепта, но дополнительные сведения вы сможете найти в [PEP 3118](#). Также обязательно стоит прочесть раздел [документации Cython](#) по «типовизированным представлениям памяти» (typed memoryviews).

15.12. Превращение указателя на функцию в вызываемый объект

Задача

У вас есть каким-то образом полученный адрес скомпилированной функции в памяти, но вы хотите превратить его в вызываемый объект Python, который можно будет использовать в качестве функции расширения.

Решение

Модуль `ctypes` может быть использован для создания вызываемых объектов Python, которые являются обёртками над произвольными адресами в памяти. Следующий пример показывает, как получить «сырой» низкоуровневый адрес функции на C и превратить его в вызываемый объект:

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary(None)
>>> # Получить адрес sin() из библиотеки C math
>>> addr = ctypes.cast(lib.sin, ctypes.c_void_p).value
>>> addr
140735505915760

>>> # Превратить адрес в вызываемую функцию
>>> funcctype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
>>> func = funcctype(addr)
>>> func
<CFunctionType object at 0x1006816d0>

>>> # Call the resulting function
>>> func(2)
0.9092974268256817
>>> func(0)
0.0
>>>
```

Обсуждение

Чтобы создать вызываемый объект, вы должны создать экземпляр `CFUNCTYPE`. Первый аргумент `CFUNCTYPE()` — это тип возвращаемого объекта. Последующие аргументы — это типы аргументов. Когда вы определили тип функции, вы оборачиваете её вокруг целочисленного адреса в памяти, чтобы создать вызываемый объект. Получившийся объект используется как обычная функция с доступом через `ctypes`.

Этот рецепт может показаться достаточно загадочным и низкоуровневым. Однако в программах и библиотеках всё чаще используют продвинутые приёмы генерации кода типа just-in-time-компиляции (это можно встретить в таких библиотеках, как LLVM).

Вот простой пример, который использует [расширение LLVMpy](#) для создания небольшой составной функции, получения функционального указателя на неё и превращения в вызываемый объект Python:

```
>>> from llvm.core import Module, Function, Type, Builder
```

```
>>> mod = Module.new('example')
>>> f = Function.new(mod, Type.function(Type.double(), \
                                         [Type.double(), Type.double()], False), 'foo')
>>> block = f.append_basic_block('entry')
>>> builder = Builder.new(block)
>>> x2 = builder.fmul(f.args[0],f.args[0])
>>> y2 = builder.fmul(f.args[1],f.args[1])
>>> r = builder.fadd(x2,y2)
>>> builder.ret(r)
<llvm.core.Instruction object at 0x10078e990>
>>> from llvm.ee import ExecutionEngine
>>> engine = ExecutionEngine.new(mod)
>>> ptr = engine.get_pointer_to_function(f)
>>> ptr
4325863440
>>> foo = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double, ctypes.c_double)

>>> # Вызываем получившуюся функцию
>>> foo(2,3)
13.0
>>> foo(4,5)
41.0
>>> foo(1,2)
5.0
>>>
```

Не стоит говорить, что любая ошибка на этом уровне вызовет ужасную смерть интерпретатора Python. Помните, что вы напрямую работаете с адресами памяти компьютера и нативным машинным кодом, а не с функциями Python.

15.13. Передача NULL-терминированных строк библиотекам на С

Задача

Вы пишете модуль расширения, который должен передавать NULL-терминированные строки в библиотеку на С. Однако вы не совсем уверены, как сделать это с помощью реализации строк Unicode в Python.

Решение

Многие библиотеки на С включают функции, которые работают с NULL-

терминированными строками, объявленными как type `char *`. Рассмотрим следующую функцию на С, которую мы будем использовать для целей демонстрации и тестирования:

```
void print_chars(char *s) {
    while (*s) {
        printf("%2x ", (unsigned char) *s);
        s++;
    }
    printf("\n");
}
```

Эта функция просто выводит шестнадцатеричное представление отдельных символов, так что передаваемые строки легко отлаживать. Например:

```
print_chars("Hello"); // Outputs: 48 65 6c 6c 6f
```

Для вызова такой функции на С из Python вы можете выбрать один из нескольких способов. Во-первых, вы можете ограничить её работой только с байтами, используя код преобразования "y" в функции `PyArg_ParseTuple()`:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;

    if (!PyArg_ParseTuple(args, "y", &s)) {
        return NULL;
    }
    print_chars(s);
    Py_RETURN_NONE;
}
```

Получившаяся функция работает так, как показано ниже. Внимательно понаблюдайте, как отвергаются байты с вставленными байтами NULL и строки Unicode:

```
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be bytes without null bytes, not bytes
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
```

```
>>>
```

Если вы хотите вместо этого передать строки Unicode, используйте код форматирования "s" в *PyArg_ParseTuple()*:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;

    if (!PyArg_ParseTuple(args, "s", &s)) {
        return NULL;
    }
    print_chars(s);
    Py_RETURN_NONE;
}
```

При использовании она автоматически преобразует все строки в NULL-терминированные в кодировке UTF-8. Например:

```
>>> print_chars('Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars('Spicy Jalape\u00f1o') # Note: UTF-8 encoding
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_chars('Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str without null characters, not str
>>> print_chars(b'Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>>
```

Если по какой-то причине вы работаете напрямую с *PyObject* * и не можете использовать *PyArg_ParseTuple()*, следующий пример кода показывает, как вы можете проверить и извлечь подходящую ссылку *char* * из байтового или строкового объекта:

```
/* Как-то полученный какой-то объект Python */
PyObject *obj;

/* Преобразование из байтов */
{
    char *s;
    s = PyBytes_AsString(o);
    if (!s) {
        return NULL; /* TypeError уже возбуждено */
    }
```

```

    }
    print_chars(s);
}

/* Преобразование в байты UTF-8 из строки */
{
    PyObject *bytes;
    char *s;
    if (!PyUnicode_Check(obj)) {
        PyErr_SetString(PyExc_TypeError, "Expected string");
        return NULL;
    }
    bytes = PyUnicode_AsUTF8String(obj);
    s = PyBytes_AsString(bytes);
    print_chars(s);
    Py_DECREF(bytes);
}

```

Оба показанных преобразования гарантируют получение NULL-терминированных данных, но они не проверяют наличие вставленных NULL-байтов где-то ещё в строке. Так что вам придётся проверить самостоятельно, если есть необходимость.

Обсуждение

При любой возможности вы должны избегать написания кода, который опирается на NULL-терминированные строки, поскольку в Python нет такого требования. Почти всегда будет лучше обрабатывать строки, используя комбинацию указателя и размера. Несмотря на это, иногда вам нужно работать с легаси-кодом на С, что не оставляет выбора.

Хотя его и легко использовать, с кодом форматирования "s" в `PyArg_ParseTuple()` вы получаете скрытый оверхед по памяти, который легко проглядеть. Когда вы пишете код, который использует такое преобразование, строка в UTF-8 создаётся и навсегда прикрепляется к изначальному объекту строки. Если оригинальная строка содержит символы не из ASCII, это увеличит размер строки до тех пор, пока она не будет уничтожена сборщиком мусора. Например:

```

>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)      # Передаем строку
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f

```

```
>>> sys.getsizeof(s)      # Заметьте, что размер увеличился
103
>>>
```

Если рост потребления памяти является проблемой, вам стоит переписать ваш код расширения на С, чтобы он использовал функцию `PyUnicode_AsUTF8String()`:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    PyObject *o, *bytes;
    char *s;

    if (!PyArg_ParseTuple(args, "U", &o)) {
        return NULL;
    }
    bytes = PyUnicode_AsUTF8String(o);
    s = PyBytes_AsString(bytes);
    print_chars(s);
    Py_DECREF(bytes);
    Py_RETURN_NONE;
}
```

С этой модификацией строка в кодировке UTF-8 создаётся при необходимости, но после использования выбрасывается. Вот изменённое поведение:

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
87
>>>
```

Если вы пытаетесь передать NULL-терминированные строки функциям, обёрнутым с помощью `ctypes`, то обратите внимание, что `ctypes` пропускает только байты — и не проверяет их на вставленные NULL-байты. Например:

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary("./libsample.so")
>>> print_chars = lib.print_chars
>>> print_chars.argtypes = (ctypes.c_char_p,)
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
```

```
>>> print_chars(b'Hello\x00World')
48 65 6c 6c 6f
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <class 'TypeError': wrong type
>>>
```

Если вы хотите передать строку вместо байтов, вам нужно выполнить ручное кодирование в UTF-8. Например:

```
>>> print_chars('Hello World'.encode('utf-8'))
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>>
```

Для других инструментов работы с расширениями (например, Swig, Cython) требуется внимательное изучение вопроса, если вы захотите использовать их для передачи строк в код на С.

15.14. Передача строк Unicode в библиотеки на С

Задача

Вы пишете модуль расширения, который должен передавать строку Python в функцию библиотеки на С, которая может уметь или не уметь правильно работать с Unicode.

Решение

В рассматриваемой задаче много проблемных мест, но основное — это то, что существующие библиотеки на С не понимают нативное представление Unicode, используемое Python. Поэтому вам нужно преобразовать строку Python в форму, которую легче поймут библиотеки на С.

Для демонстрационных целей покажем две функции на С, которые работают над строковыми данными и выводят их (для отладки и экспериментов). Одна использует байты, предоставленные в форме `char * , int`, тогда как другая использует широкие символы в форме `wchar_t * , int`:

```

void print_chars(char *s, int len) {
    int n = 0;
    while (n < len) {
        printf("%2x ", (unsigned char) s[n]);
        n++;
    }
    printf("\n");
}

void print_wchars(wchar_t *s, int len) {
    int n = 0;
    while (n < len) {
        printf("%x ", s[n]);
        n++;
    }
    printf("\n");
}

```

Для байт-ориентированной функции *print_chars()* вам нужно преобразовать строки Python в подходящую байтовую кодировку, такую как UTF-8. Вот пример функции расширения, которая это делает:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "s#", &s, &len)) {
        return NULL;
    }
    print_chars(s, len);
    Py_RETURN_NONE;
}

```

Для библиотечных функций, которые работают с нативным машинным типом *wchar_t*, вы можете написать такой код расширения:

```

static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    wchar_t *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "u#", &s, &len)) {
        return NULL;
    }
    print_wchars(s, len);
    Py_RETURN_NONE;
}

```

Вот интерактивный сеанс, который демонстрирует работу этих функций:

```
>>> s = 'Spicy Jalape\u00f1o'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>>
```

Поналюдайте, как байт-ориентированная функция *print_chars()* получает закодированные в UTF-8 данные, в то время, как *print_wchars()* получает значения кодовых точек Unicode (Unicode code point values).

Обсуждение

Перед рассмотрением этого рецепта вы должны сначала изучить природу библиотеки на С, к которой вы обращаетесь. Многим библиотекам на С имеет смысл передавать байты, а не строки. Чтобы сделать это, используйте код для преобразования:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;
    Py_ssize_t len;

    /* Принимает байты, байтовый массив или другой байтоподобный объект */
    if (!PyArg_ParseTuple(args, "y#", &s, &len)) {
        return NULL;
    }
    print_chars(s, len);
    Py_RETURN_NONE;
}
```

Если вы решите, что всё равно хотите передавать строки, то вам нужно знать, что Python 3 использует адаптируемое представление строк, которое не так уж прямолинейно отображается напрямую в библиотеки на С с помощью стандартных типов *char ** или *wchar_t **. За подробностями обратитесь к [PEP 393](#). Поэтому для представления строковых данных в С практически всегда нужно выполнить какое-то преобразование. Коды форматирования *s#* и *u#* для функции *PyArg_ParseTuple()* безопасно выполняют такие преобразования.

Потенциальный недостаток таких преобразований — перманентное увеличение размера изначального строкового объекта. Когда

преобразование сделано, копия преобразованных данных сохраняется и прикрепляется к изначальному объекту строки для целей переиспользования. Вы можете понаблюдать за этим эффектом:

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
103
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>> sys.getsizeof(s)
163
>>>
```

Для небольших объёмов строковых данных это, возможно, не так уж важно, но если вы выполняете обработку текстов в промышленных масштабах, то, вероятно, захотите избавиться от оверхеда. Вот альтернативная реализация первой функции расширения, которая исключает эти проблемы с неэффективным использованием памяти:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    PyObject *obj, *bytes;
    char *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }
    bytes = PyUnicode_AsUTF8String(obj);
    PyBytes_AsStringAndSize(bytes, &s, &len);
    print_chars(s, len);
    Py_DECREF(bytes);
    Py_RETURN_NONE;
}
```

Избежать оверхеда по памяти при обработке *wchar_t* намного сложнее. Внутри Python хранит строки, используя наиболее эффективное из возможных представлений. Например, строки, содержащие только ASCII-символы, хранятся как массивы байтов, тогда как строки, содержащие символы в диапазоне от U+0000 до U+FFFF используют двухбайтовое представление. Поскольку единого представления данных нет, вы не можете просто

приводить внутренний массив к `wchar_t *` и надеяться, что всё будет работать. Вместо этого нужно создать массив `wchar_t` и скопировать в него текст. Код форматирования "`u#`" для `PyArg_ParseTuple()` сделает это за вас, но придётся заплатить снижением эффективности (она прикрепляет получившуюся копию к объекту строки).

Если вы хотите избежать этого долговременного оверхеда по памяти, ваш единственный выход — скопировать данные в Unicode во временный массив, передать его в библиотечную функцию на С, а затем dealloцировать массив. Вот возможный способ реализации:

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    PyObject *obj;
    wchar_t *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }
    if ((s = PyUnicode_AsWideCharString(obj, &len)) == NULL) {
        return NULL;
    }
    print_wchars(s, len);
    PyMem_Free(s);
    Py_RETURN_NONE;
}
```

В этой реализации `PyUnicode_AsWideCharString()` создаёт временный буфер символов `wchar_t` и копирует в него данные. Этот буфер передаётся в С и затем высвобождается. Когда писалась эта книга, с этим поведением был связан возможный баг, что описано на [странице багов Python](#).

Если по какой-то причине вы знаете, что библиотека на С принимает данные в байтовой кодировке, отличной от UTF-8, вы можете заставить Python выполнить нужное преобразование, используя код расширения:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s = 0;
    int len;
    if (!PyArg_ParseTuple(args, "es#", "encoding-name", &s, &len)) {
        return NULL;
    }
    print_chars(s, len);
    PyMem_Free(s);
    Py_RETURN_NONE;
```

```
}
```

И последнее: если вы хотите напрямую работать с символами в строке Unicode, вот пример, который демонстрирует низкоуровневый доступ:

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    PyObject *obj;
    int n, len;
    int kind;
    void *data;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }
    if (PyUnicode_READY(obj) < 0) {
        return NULL;
    }

    len = PyUnicode_GET_LENGTH(obj);
    kind = PyUnicode_KIND(obj);
    data = PyUnicode_DATA(obj);

    for (n = 0; n < len; n++) {
        Py_UCS4 ch = PyUnicode_READ(kind, data, n);
        printf("%x ", ch);
    }
    printf("\n");
    Py_RETURN_NONE;
}
```

В этом коде макросы `PyUnicode_KIND()` и `PyUnicode_DATA()` относятся к хранилищу Unicode переменной ширины, как описано в [PEP 393](#). Переменная `kind` кодирует информацию о лежащем в основе хранилище (8 бит, 16 бит или 32 бит), а `data` указывает буфер. На самом деле вам не нужно ничего делать с этими значениями, пока вы передаёте их в макрос `PyUnicode_READ()`, когда извлекаете символы.

Пара слов напоследок: при передаче строк Unicode из Python в C, вы, вероятно, попытаетесь максимально упростить этот процесс. Если у вас есть выбор между такими кодировками, как UTF-8 и широкие символы, выбирайте UTF-8. Поддержка UTF-8 намного более распространена, эта кодировка менее подвержена ошибкам и лучше поддерживается интерпретатором. И обязательно почитайте [документацию по работе с Unicode](#).

15.15. Преобразование строк С в Python

Задача

Вы хотите преобразовать строки из С в байтовый или строковый объект Python.

Решение

Для строк С, представленных как пара `char *`, вы должны выбрать, хотите ли вы представить строку, как сырую байтовую строку или строку Unicode.

Байтовые объекты могут быть созданы с помощью `Py_BuildValue()`:

```
char *s;      /* Указатель на строковые данные С */
int len;     /* Длина данных */

/* Make a bytes object */
PyObject *obj = Py_BuildValue("y#", s, len);
```

Если вы хотите создать строку Unicode и знаете, что `s` указывает на данные, закодированные в UTF-8, вы можете сделать так:

```
PyObject *obj = Py_BuildValue("s#", s, len);
```

Если `s` представлено в какой-то другой известной кодировке, вы можете создать строку с помощью `PyUnicode_Decode()`:

```
PyObject *obj = PyUnicode_Decode(s, len, "encoding", "errors");

/* Примеры */
obj = PyUnicode_Decode(s, len, "latin-1", "strict");
obj = PyUnicode_Decode(s, len, "ascii", "ignore");
```

Если у вас есть широкая строка, представленная как `wchar_t *`, `len` pair, то есть несколько способов. Во-первых, вы можете использовать `Py_BuildValue()`:

```
wchar_t *w;      /* Стока широких символов */
int len;        /* Длина */

PyObject *obj = Py_BuildValue("u#", w, len);
```

Или же вы можете использовать `PyUnicode_FromWideChar()`:

```
PyObject *obj = PyUnicode_FromWideChar(w, len);
```

Для строк из широких символов не выполняется интерпретация символьных данных — предполагается, что это сырье точки кода Unicode (raw Unicode code points), которые напрямую конвертируются в Python.

Обсуждение

Конвертирование строк из C в Python подчиняется тем же принципам, что и ввод-вывод. А именно, данные из C должны быть явно декодированы в строку в соответствии с каким-то кодеком. Распространённые кодировки включают ASCII, Latin-1 и UTF-8. Если вы не до конца уверены в выборе кодировки или работаете с бинарными данными, то, вероятно, лучше кодировать не в строки, а в байты.

При создании объекта Python всегда копирует предоставленные строковые данные. Если это нужно, то вы можете высвободить строку C после завершения. Также, для повышения надежности, вам стоит попытаться создавать строки с помощью указателя и размера, а не полагаться на NULL-терминированные данные.

15.16. Работа со строками С в сомнительной кодировке

Задача

Вы преобразовываете строки туда и обратно между C и Python, но кодировка С сомнительна или неизвестна. Возможно, например, что данные С должны быть в UTF-8, но строгого принуждения к этому нет. Вы хотели бы написать код, который может обрабатывать искажённые данные аккуратным способом, который не обрушит Python и не разрушит строковые данные в процессе.

Решение

Вот некоторые данные С и функция, которые демонстрируют суть проблемы:

```
/* Какие-то сомнительные строковые данные (искаженные UTF-8) */
```

```

const char *sdata = "Spicy Jalape\xc3\xb1o\xae";
int slen = 16;

/* Вывод символьных данных */
void print_chars(char *s, int len) {
    int n = 0;
    while (n < len) {
        printf("%2x ", (unsigned char) s[n]);
        n++;
    }
    printf("\n");
}

```

В этом коде строка `sdata` содержит смесь UTF-8 и искажённых данных. Тем не менее, если пользователь вызывает в C `print_chars(sdata, slen)`, то всё отлично работает.

Теперь предположим, что вы хотите преобразовать содержимое `sdata` в строку Python. Также предположим, что позже вы захотите передать эту строку в функцию `print_chars()` через расширение. Вот как сделать это, в точности сохранив изначальные данные — даже с учётом, что есть проблемы с кодировкой:

```

/* Возврат строки С обратно в Python */
static PyObject *py_retstr(PyObject *self, PyObject *args) {
    if (!PyArg_ParseTuple(args, "")) {
        return NULL;
    }
    return PyUnicode_Decode(sdata, slen, "utf-8", "surrogateescape");
}

/* Обёртка для функции print_chars() */
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    PyObject *obj, *bytes;
    char *s = 0;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }

    if ((bytes = PyUnicode_AsEncodedString(obj, "utf-8", "surrogateescape"
                                           == NULL) {
        return NULL;
    }
    PyBytes_AsStringAndSize(bytes, &s, &len);
    print_chars(s, len);
    Py_DECREF(bytes);
    Py_RETURN_NONE;
}

```

```
}
```

Если вы попробуете вызвать эти функции из Python, то произойдёт вот это:

```
>>> s = retstr()
>>> s
'Spicy Jalapeño\udcae'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f ae
>>>
```

Внимательный наблюдатель обнаружит, что искажённая строка кодируется в строку Python без ошибок, а затем передаётся обратно в С, превращаясь обратно в байтовую строку, которая кодируется теми же байтами, что и изначальная строка С.

Обсуждение

Этот рецепт относится к тонкому, но потенциально раздражающему моменту, касающемуся обработки строк в модулях расширения. А именно, к тому факту, что строки С в расширениях могут не следовать строгим правилам кодирования/декодирования Unicode, соблюдения которых в обычном случае ожидает Python. Поэтому возможно, чтобы какие-то искажённые данные С будут переданы в Python. Хороший пример — строки С, ассоциированные с низкоуровневыми системными вызовами, такими как имена файлов.

Например, проблема может возникнуть, когда системный вызов возвращает битую строку обратно в интерпретатор, который не может её правильно декодировать.

В обычном случае ошибки Unicode часто обрабатываются специальными правилами обработки ошибок, такими как *strict*, *ignore*, *replace* или похожими. Однако недостаток этих правил в том, что они безвозвратно уничтожают изначальное содержимое строки. Например, если искажённые данные из примера будут декодированы с помощью этих правил, то вы получите такие результаты:

```
>>> raw = b'Spicy Jalape\xc3\xb1o\xae'
>>> raw.decode('utf-8', 'ignore')
'Spicy Jalapeño'
>>> raw.decode('utf-8', 'replace')
'Spicy Jalapeño?'
>>>
```

Правило обработки ошибок `surrogateescape` принимает все недекодируемые байты и превращает их в нижнюю половину (low-half) суррогатной пары (\udcXX, где XX — это значение сырого байта). Например:

```
>>> raw.decode('utf-8', 'surrogateescape')
'Spicy Jalapeño\udcae'
>>>
```

Изолированные нижние символы-суррогаты, такие как \udcae, никогда не появляются в валидном Unicode. Так что эта строка технически является недопустимым представлением. На самом деле, если вы попытаетесь передать её в функции, которые выполняют вывод, то получите ошибки кодировки:

```
>>> s = raw.decode('utf-8', 'surrogateescape')
>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udcae'
in position 14: surrogates not allowed
>>>
```

Однако главная цель допустимости этих суррогатных экранирований в том, чтобы позволитьискажённым строкам передаваться из C в Python (и обратно в C) без потерь данных. Когда строка снова кодируется с помощью `surrogateescape`, суррогатные символы превращаются обратно в свои изначальные байты. Например:

```
>>> s
'Spicy Jalapeño\udcae'
>>> s.encode('utf-8', 'surrogateescape')
b'Spicy Jalape\xc3\xb1o\xae'
>>>
```

В качестве общего правила: вероятно, лучше при любой возможности избегать суррогатной кодировки. Ваш код будет намного более надёжным, если использует правильные кодировки. Однако иногда возникают ситуации, когда у вас просто нет контроля над кодировкой данных, и вы не можете игнорировать или заменять плохие данные, поскольку другие функции могут в них нуждаться. Этот рецепт показывает, как решить подобные проблемы.

Последнее замечание: многие системно-ориентированные функции Python.

особенно те, что работают с именами файлов, переменными окружения и аргументами командной строки, используют суррогатную кодировку.

Например, если вы используете функцию типа `os.listdir()` в каталоге, содержащем недекодируемое имя файла, она вернёт строку с суррогатными экранированными последовательностями. См. [рецепт 5.15](#).

[PEP 383](#) содержит дополнительные сведения о проблеме, которая рассмотрена в этом рецепте, а также об обработке ошибок `surrogateescape`.

15.17. Передача имён файлов в расширения на C

Задача

Вам нужно передать имена файлов в функции библиотеки на C, но вы хотите убедиться, что имена правильно закодированы в соответствии с ожидаемой системной кодировкой имён файлов.

Решение

Чтобы написать функцию расширения, которая принимает имя файла, используйте такой код:

```
static PyObject *py_get_filename(PyObject *self, PyObject *args) {
    PyObject *bytes;
    char *filename;
    Py_ssize_t len;
    if (!PyArg_ParseTuple(args, "O&", PyUnicode_FSConverter, &bytes)) {
        return NULL;
    }
    PyBytes_AsStringAndSize(bytes, &filename, &len);
    /* Используем имя файла */
    ...
    /* Очистка и возврат */
    Py_DECREF(bytes)
    Py_RETURN_NONE;
}
```

Если у вас уже есть `PyObject *`, который вы хотите преобразовать в имя файла, используйте такой код:

```
PyObject *obj; /* Объект с именем файла */
PyObject *bytes;
char *filename;
Py_ssize_t len;

bytes = PyUnicode_EncodeFSDefault(obj);
PyBytes_AsStringAndSize(bytes, &filename, &len);
/* Используем имя файла */
...

/* Подчистка */
Py_DECREF(bytes);
```

Если вам нужно вернуть имя файла обратно в Python, используйте следующий код:

```
/* Превращение имени файла в объект Python */

char *filename; /* Уже установлено */
int filename_len; /* Уже установлено */

PyObject *obj = PyUnicode_DecodeFSDefaultAndSize(filename, filename_len)
```

Обсуждение

Работа с именами файлов переносимым образом — это сложная задача, которую лучше переложить на Python. Если вы используете этот рецепт в вашем коде расширения, имена файлов будут обрабатываться способом, который соответствует принципам обработки в Python. Это включает кодирование/декодирование байтов, решение проблем с некорректными символами, суррогатные экранированные последовательности и другие осложнения.

15.18. Передача открытых файлов в расширения на С

Задача

У вас есть открытый файловый объект в Python, но вы хотите передать его коду расширения на С, который сможет использовать этот файл.

Решение

Чтобы преобразовать файл в целочисленный файловый дескриптор, используйте `PyFile_FromFd()`:

```
PyObject *fobj;      /* Файловый объект (полученный каким-то образом) */
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0) {
    return NULL;
}
```

Получившийся файловый дескриптор получен путём вызова метода `fileno()` на объекте `fobj`. Так что это должно работать с любым объектом, который показывает наружу файловый дескриптор совместимым способом (например, файл, сокет и т.п.)

Когда вы получили файловый дескриптор, вы можете передать его в различные низкоуровневые функции на С, которые могут работать с файлами.

Если вам нужно преобразовать целочисленный файловый дескриптор обратно в объект Python, используйте `PyFile_FromFd()`:

```
int fd;      /* Существующий файловый дескриптор (уже открытый) */
PyObject *fobj = PyFile_FromFd(fd, "filename", "r", -1, NULL, NULL, NULL, 1);
```

Аргументы `PyFile_FromFd()` отражают те, что используются функцией `open()`. Значения `NULL` просто показывают, что использованы значения по умолчанию для аргументов `encoding`, `errors` и `newline`.

Обсуждение

Если вы передаёте файловые объекты из Python в C, то вам придётся разобраться с несколькими тонкими моментами. Во-первых, Python выполняет собственную буферизацию ввода-вывода через модуль `io`. Перед тем, как передавать любой файловый дескриптор в C, вы сначала должны очистить (`flush`) буфера ввода-вывода ассоциированных файловых объектов. В противном случае данные в потоке файла у вас могут появиться в неупорядоченном виде.

Во-вторых, нужно внимательно следить за владением файлами и, в

особенности, за закрытием файлов. Если файловый дескриптор передан в C, но всё еще используется в Python, вам нужно убедиться, что C случайно не закроет этот файл. Похожим образом, если файловый дескриптор превращён в файловый объект Python, вам нужно чётко понимать, на ком лежит ответственность за его закрытие. Последний аргумент *PyFile_FromFd()* установлен на 1, чтобы обозначить, что закрыть файл должен Python.

Если вам нужно создать другой тип файлового объекта, такой как объект FILE * из стандартной библиотеки ввода-вывода C с использованием функции *fdopen()*, вам следует быть особенно осторожным. Если вы это сделаете, то возникнут два полностью отдельных слоя буферизации ввода-вывода в стеке ввода-вывода (один из модуля *io* Python, второй — из C *stdio*). Операции типа *fclose()* в C также могут ненароком закрыть файл от будущего использования в Python. Если у вас есть выбор, то, вероятно, лучше заставить код расширения работать с низкоуровневыми целочисленными файловыми дескрипторами, а не использовать высокоуровневую абстракцию наподобие предоставляемой .

15.19. Чтение файлоподобных объектов из С

Задача

Вы хотите написать код расширения на С, который будет потреблять данные из любого файлоподобного объекта Python (например, обычных файлов, объектов *StringIO* и т.п.)

Решение

Чтобы потреблять данные из файлоподобного объекта, вам нужно раз за разом вызывать его метод *read()* и предпринимать шаги для правильного декодирования получившихся данных.

Вот пример функции расширения на С, которая просто потребляет все данные из файлоподобного объекта и сбрасывает их в стандартный поток вывода, чтобы вы могли их увидеть:

```
#define CHUNK_SIZE 8192
/* Потребляем файлоподобный объект и записываем байты в stdout */
```

```
static PyObject *py_consume_file(PyObject *self, PyObject *args) {
    PyObject *obj;
    PyObject *read_meth;
    PyObject *result = NULL;
    PyObject *read_args;

    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }

    /* Получить метод чтения переданного объекта */
    if ((read_meth = PyObject_GetAttrString(obj, "read")) == NULL) {
        return NULL;
    }

    /* Построение списка аргументов для read() */
    read_args = Py_BuildValue("(i)", CHUNK_SIZE);
    while (1) {
        PyObject *data;
        PyObject *enc_data;
        char *buf;
        Py_ssize_t len;

        /* Вызов read() */
        if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL)
            goto final;
    }

    /* Проверка EOF */
    if (PySequence_Length(data) == 0) {
        Py_DECREF(data);
        break;
    }

    /* Кодируем Unicode в Bytes для C */
    if ((enc_data=PyUnicode_AsEncodedString(data,"utf-8","strict"))=
        Py_DECREF(data));
        goto final;
    }

    /* Извлекаем данные буфера */
    PyBytes_AsStringAndSize(enc_data, &buf, &len);

    /* Записываем в stdout (заменить чем-то более полезным) */
    write(1, buf, len);

    /* Подчистка */
    Py_DECREF(enc_data);
    Py_DECREF(data);
}

result = Py_BuildValue("");
```

```
final:  
    /* Подчистка */  
    Py_DECREF(read_meth);  
    Py_DECREF(read_args);  
    return result;  
}
```

Чтобы проверить этот код, попробуйте создать файлоподобный объект типа экземпляра *StringIO* и передать его:

```
>>> import io  
>>> f = io.StringIO('Hello\nWorld\n')  
>>> import sample  
>>> sample.consume_file(f)  
Hello  
World  
>>>
```

Обсуждение

В отличие от обычных системных файлов, файлоподобные объекты необязательно построены на базе низкоуровневого файлового дескриптора. Поэтому вы не можете использовать обычные функции библиотек С для доступа к ним. Вместо этого вам нужен Python C API, с помощью которого вы можете работать с файлоподобным объектом во многом так же, как вы делаете это в Python.

В решении метод *read()* извлечён из переданного объекта. Список аргументов строится и затем раз за разом передаётся в *PyObject_Call()*, чтобы вызвать метод. Чтобы обнаружить конец файла (EOF), используется *PySequence_Length()* — чтобы посмотреть, имеет ли возвращённый результат нулевую длину.

При всех операциях ввода-вывода вам нужно думать над лежащей в основе кодировкой и различием между байтами и Unicode. Этот рецепт показывает, как прочесть файл в текстовом режиме и декодировать получившийся текст в байтовую кодировку, которая может быть использована в С. Если вы хотите прочесть файл в бинарном режиме, нужно внести в код лишь косметические изменения. Например:

```
...  
/* Вызов read() */
```

```

if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
    goto final;
}

/* Проверка на EOF */
if (PySequence_Length(data) == 0) {
    Py_DECREF(data);
    break;
}

if (!PyBytes_Check(data)) {
    Py_DECREF(data);
    PyErr_SetString(PyExc_IOError, "File must be in binary mode");
    goto final;
}

/* Извлекаем данные буфера */
PyBytes_AsStringAndSize(data, &buf, &len);
...

```

Самая сложная часть этого рецепта касается правильного управления памятью. При работе с переменными `PyObject *` нужно внимательно следить за управлением подсчётом ссылок и очисткой значений после того, как они уже не требуются. Различные вызовы `Py_DECREF()` занимаются как раз этим.

Этот рецепт написан так, чтобы иметь общее назначение — его можно адаптировать к другим операциям, таким как запись. Например, чтобы записать данные, просто получите метод `write()` из файлоподобного объекта, преобразуйте данные в подходящий объект Python (байты или Unicode) и вызовите метод, чтобы записать его в файл.

Наконец, хотя файлоподобные объекты часто предоставляют другие методы (например, `readline()`, `read_into()`), вероятно, лучше будет придерживаться базовых методов `read()` и `write()` для максимальной переносимости. Придерживаться максимальной простоты — хорошая политика при работе с расширениями на С.

15.20. Потребление итерируемого объекта из С

Задача

Вы хотите написать код расширения на С, который потребляет элементы из

любого итерируемого объекта, такого как список, кортеж, файл или генератор.

Решение

Вот пример расширения на C, который показывает, как потреблять элементы из итерируемого объекта:

```
static PyObject *py_consume_iterable(PyObject *self, PyObject *args) {
    PyObject *obj;
    PyObject *iter;
    PyObject *item;

    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }

    if ((iter = PyObject_GetIter(obj)) == NULL) {
        return NULL;
    }

    while ((item = PyIter_Next(iter)) != NULL) {
        /* Используем элемент */
        ...
        Py_DECREF(item);
    }
    Py_DECREF(iter);
    return Py_BuildValue("");
}
```

Обсуждение

Код в этом рецепте отражает похожий код на Python. Вызов `PyObject_GetIter()` — это то же самое, что и вызов `iter()` для получения итератора. Функция `PyIter_Next()` вызывает следующий метод на итераторе, возвращая следующий элемент или `NULL`, если элементов больше нет. Убедитесь, что вы осторожно управляете памятью — `Py_DECREF()` нужно вызывать и на производимых элементах, и на самом объекте итератора для избежания утечек памяти.

15.21. Диагностика ошибок сегментации

Задача

Интерпретатор падает с ошибкой сегментации, ошибкой шины, нарушением доступа или другим фатальным событием. Вы хотели бы получить трейсбэк Python, который покажет вам, где ваша программа наткнулась на точку отказа.

Решение

Вам поможет модуль *faulthandler*. Включите следующий код в вашу программу:

```
import faulthandler  
faulthandler.enable()
```

В качестве альтернативы вы можете запустить Python с опцией *-Xfaulthandler*:

```
bash % python3 -Xfaulthandler program.py
```

И последнее: вы можете установить переменную окружения *PYTHONFAULTHANDLER*.

Когда *faulthandler* включен, фатальные ошибки в расширениях на С будут приводить к тому, что будет выводиться трейсбэк Python. Например:

```
Fatal Python error: Segmentation fault  
  
Current thread 0x00007fff71106cc0:  
  File "example.py", line 6 in foo  
  File "example.py", line 10 in bar  
  File "example.py", line 14 in spam  
  File "example.py", line 19 in <module>  
Segmentation fault
```

Хотя этот приём не покажет вам, где в коде на С всё пошло не так, но, по крайней мере, у вас будет информация из Python.

Обсуждение

faulthandler покажет вам трейсбэк стека кода Python, исполнявшегося в момент возникновения ошибки. По меньшей мере вы узнаете, какая функция расширения высшего уровня была вызвана. С помощью *pdb* или другого

дебаггера Python вы можете исследовать поток выполнения кода Python, который привёл к ошибке.

faulthandler не расскажет вам ничего о причинах ошибки в C. Для этого вам нужно использовать традиционный дебаггер C, такой как *gdb*. Однако информация из трейсбэка *faulthandler* может навести вас на мысль о том, где нужно копать.

Стоит отметить, что некоторые типы ошибок в C не так-то и легко исправить. Например, если расширение на C замусорит стек или кучу (heap) программы, это может вывести *faulthandler* из рабочего состояния, и вы просто-напросто не получите никакого вывода (только падение программы). Очевидно, что в каждом конкретном случае поведение может сильно отличаться.