Ч. Северенс

Введение в программирование на Python

ИНТУИТ

НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

ИНТУИТ

НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Введение в программирование на Python

2-е издание, исправленное

Северенс Ч.

Национальный Открытый Университет "ИНТУИТ"
2016

Введение в программирование на Python

2-е издание, исправленное

Северенс Ч.

Национальный Открытый Университет "ИНТУИТ"
2016

Введение в программирование на Python/ Ч. Северенс - М.: Национальный Открытый Университет "ИНТУИТ", 2016

Вводный курс по программированию дает представление о базовых понятиях структурного программирования (данных, операциях, переменных, ветвлениях в программе, циклах и функциях).
Python обладает рядом преимуществ перед другими языками для начинающих изучать программирование, прежде всего благодаря ясности кода и быстроты реализации.

Почему следует научиться писать программы?

Видео

Компьютеры способны быстро решать задачи, которые являются трудоемкими для человека. Например, подсчет частоты встречаемости слов в тексте и определение слов, которые встречаются чаще всего. Человек способен решить такую задачу, но это скучная и однообразная работа. Компьютер (или PDA - персональный цифровой помощник) справляется с ней быстро:

```
python words.py
Enter file:words.txt
to 16
```

Это пример выполнения программы, которая в дальнейшем будет разбираться более подробно.

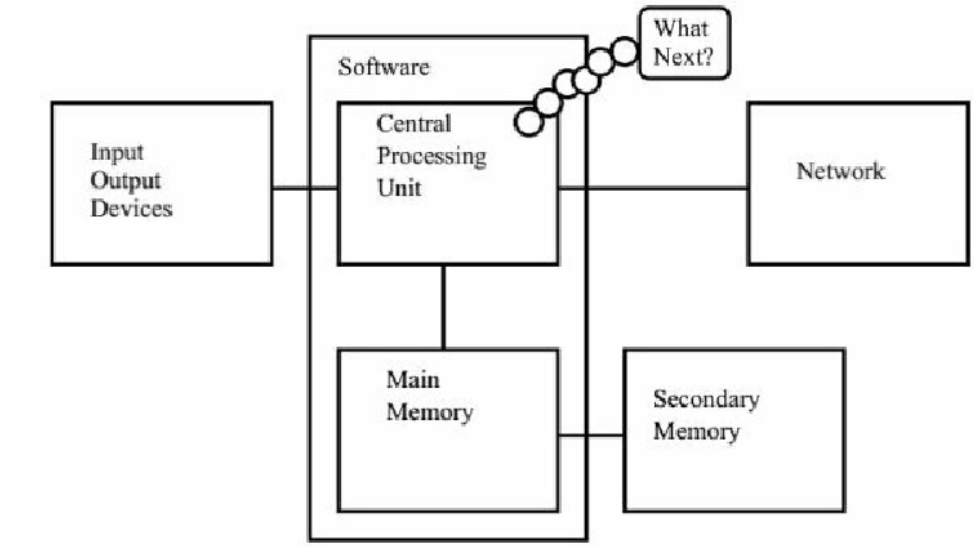
Выучив язык программирования Python, вы сможете передать решение повседневных задач помощнику (компьютеру), таким образом, вы освободите время для решения более интересных проблем.

1.1. Творчество и мотивация

Создание программ, которые будут использоваться другими людьми - это творческое занятие. Огромное количество программ на компьютере конкурируют за ваше внимание и интерес (программисты зарабатывают на этом деньги). Задачей данного пособия является не зарабатывание денег или радость конечного пользователя, а более продуктивное выполнение повседневных задач обработки данных.

1.1. Архитектура компьютера

Если бы вы разобрали компьютер или мобильный телефон и заглянули внутрь, то увидели бы следующие части:



На высоком уровне определения частей следующие:

- Центральный процессор (CPU) - часть компьютера, которая создана, чтобы постоянно спрашивать "что дальше"? Если частота вашего компьютера 3.0 ГГц, это означает, что CPU будет спрашивать "что дальше?" три миллиарда раз в секунду.

- Оперативная память (Main Memory) используется для хранения информации, которая срочно необходима CPU. Информация, хранящаяся в оперативной памяти, стирается, когда питание компьютера прекращается.

- Вторичная память (Secondary Memory) также нужна для хранения информации, но она медленнее оперативной памяти. Особенностью вторичной памяти является возможность хранить информацию после выключения питания компьютера. Примеры вторичной памяти: диски или флеш-память.

- Устройства ввода/вывода (Input and Output Devices) – наш монитор, клавиатура, мышь, микрофон, колонки и т.д. Они обеспечивают взаимодействие с компьютером.

- В наше время большинство компьютеров имеют сетевой адаптер (Network Connection) для обмена информации через сеть.

1.3. Понимание программирования

В остальной части книги, мы постараемся превратить вас в человека, который является специалистом в области программирования.

В конце вы станете программистом, возможно, не профессиональным программистом, но, по крайней мере, вы будете иметь навыки исследования проблем анализа данных/информации и разработки программ для их решения.

В известном смысле, нужно два навыка, чтобы стать программистом:

- Во-первых, необходимо знать язык программирования (в нашем случае, Python) - вам необходимо знать лексику и грамматику. В новом языке вы должны правильно строить слова и "предложения".

- Во-вторых, вам необходимо "рассказать историю". При написании истории, вы объединяете слова и фразы, передающие сюжет читателю. В программировании, наша программа - это "история", а проблема, которую вы пытаетесь решить – это сюжет.

Однажды выучив язык программирования, такой как Python, вы обнаружите, что сможете легко выучить второй язык программирования, такой как JavaScript или C++. Новый язык программирования имеет много отличий в словаре и грамматике, но как только вы приобретете навыки решения задач, они будут одинаковыми во всех языках программирования.

Вы очень быстро выучите словарь (vocabulary) и выражения (sentences) Python. Но чтобы научиться писать большие программы для решения сложных проблем, понадобится время. Мы изучим программирование подобно изучению письму. Мы начнем читать и разбирать программы, затем напишем простые программы и только потом перейдем к более сложным. В некоторой момент вы "поймаете вашу писательскую музу", самостоятельно увидите шаблоны решения проблемы и сможете написать программу. С этого момента программирование становится очень приятным и творческим процессом.

Мы начнем со словаря и структуры программы на Python. Будьте терпеливы, в первый раз вы учились читать на простых примерах.

1.4. Слова и фразы

В отличие от человеческих языков, словарь Python значительно меньше. Назовем этот словарь списком зарезервированных слов. Существуют слова, которые в Python имеют специальное значение. Когда вы встречаете эти слова в программе на Python, они имеют одно и только одно значение для Python. Позже вы будете писать программы и создавать переменные (variables) - собственные слова, которые имеют смысл для вас. У вас будет широкий выбор имен для ваших переменных, но вы не можете использовать в качестве имен переменных зарезервированные слова.

В известном смысле, когда вы дрессируете собаку, вы можете использовать специальные слова, например, "сидеть", "стоять", "принести". Когда вы общаетесь с собакой и не используете какие-либо зарезервированные слова, они только смотрят на вас, пока вы не произнесете зарезервированное слово. К примеру, если вы скажете: "Я желаю большинству людей ходить (walk) для улучшения здоровья", то собака услышит нечто подобное: "ла ла ла ла ла ла ходить (walk) ла ла ла ла ла ла". Слово ходить (walk) является зарезервированным в языке собаки.

Зарезервированные в Python слова:

```
and del for is raise assert elif from lambda return break else
global not try class except if or while continue exec import
pass yield def nally in print
```

В отличие от собаки, Python уже обучен этим словам.

Позже мы изучим зарезервированные слова и то, как они используются, а сейчас сосредоточимся на эквиваленте слову "говорить" (speak) в Python:

```
print 'Hello world!'
```

Мы написали нашу первую синтаксически правильную фразу на Python. Наша фраза начинается с зарезервированного слова print и продолжается текстовой строкой, заключенной в одиночные кавычки.

1.5. Разговаривающий с Python

Теперь, когда у нас есть слово и простая фраза, которую мы знаем на Python, необходимо научиться общению с Python для проверки наших языковых навыков.

Предварительно нам надо установить Python на компьютер и запустить в интерактивном режиме. Инструкция по установке Python в ОС Windows будет здесь: ссылка: http://pycode.ru/edu/why-python/

В интерактивном режиме получим следующее:

```
Python 1.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on
win32

Type "copyright", "credits" or "license()" for more information.
>>>
```

С помощью символов >>> интерпретатор Python спрашивает: "Что вы хотите, чтобы я сделал дальше?". Теперь Python готов с вами общаться.

Скажем, для примера вы не знали простых слов или фраз для общения с Python. Вы захотели использовать стандартные строки, которые астронавты используют, когда прилетают на другую планету и пытаются поговорить с жителями этой планеты:

```
>>> I come in peace, please take me to your leader
SyntaxError: invalid syntax
```

Это не очень хорошо. Жители планеты, скорее всего, вас поджарят и съедят на ужин. К счастью, вы захватили с собой в путешествие эту книгу.

Попробуйте еще раз:

```
>>> print 'Hello world!'
Hello world!
>>> print 'Привет, мир!'
Привет, мир!
```

Это выглядит гораздо лучше, поэтому попытайтесь сообщить еще немного:

```
>>> print 'You must be the legendary god that comes from the sky'
You must be the legendary god that comes from the sky
>>> print 'We have been waiting for you for a long time'

We have been waiting for you for a long time
>>> print 'Our legend says you will be very tasty with mustard'
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
SyntaxError: EOL while scanning string literal
>>>
```

Разговор шел хорошо, пока вы не совершили маленькую ошибку в языке Python.

На данный момент, вы также должны понимать, что Python удивительно сложный и мощный язык, он очень требователен к синтаксису, используемому для взаимодействия с ним, но Python НЕ обладает разумом. Вы беседуете сами с собой, но с использованием правильного синтаксиса.

В некотором смысле, когда вы используете написанную кем-то программу, общение осуществляется между вами и теми другими программистами, а в качестве посредника выступает Python.

И через несколько глав вы станете одним из тех программистов на Python, которые общаются с пользователями программ.

Перед тем как покинуть наш первый разговор с интерпретатором Python, вам необходимо знать, как правильно сказать "до свидания" при взаимодействии с жителями планеты Python:

```
>>> good-bye
```

```
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    good-bye
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
SyntaxError: invalid syntax
>>> quit()
```

Вы заметили, что для двух первых неудачных попыток ошибки различаются. Вторая ошибка отличается тем, что, if - зарезервированное слово и Python подумал, что мы пытаемся что-то сказать, но используем неправильный синтаксис.

Правильный способ сказать "до свидания" на Python, это ввести quit() в интерактивной строке приглашения.

1.6. Терминология: интерпретатор и компилятор

Python - высокоуровневый (high-level) язык программирования, созданный, чтобы быть относительно простым для чтения и написания программ человеком, и для компьютерного чтения и выполнения. Другие высокоуровневые языки: Java, C++, PHP, Ruby, Basic, Perl, JavaScript и множество других.

Имеющиеся аппаратные средства внутри CPU не понимают любой из вышеперечисленных языков. CPU понимает язык, который называется машинным (machine-language). Машинный язык является простым, но очень утомительным в написании, т.к. представляет собой только нули и единицы:

```
01010001110100100101010000001111
11100110000011101010010101101101
...
```

Машинный язык кажется довольно простым на поверхности, учитывая, что в нем есть только нули и единицы, но он обладает сложным синтаксисом. Поэтому очень немногие программисты когда-либо писали на машинном языке. Вместо этого мы строим различные

трансляторы (переводчики), позволяющие программистам писать на языках высокого уровня, таких как Python или JavaScript. Затем трансляторы преобразуют программы в машинный язык для исполнения их процессором.

Поскольку машинный язык привязан к аппаратному обеспечению компьютера, он не может быть перенесен (portable) на компьютер с другим типом оборудования. Программы, написанные на языках высокого уровня, можно перемещать между разными компьютерами, например, с использованием транслятора на новом компьютере, или с помощью перекомпиляции кода для создания новой версии машинного языка программы.

Трансляторы языков программирования делятся на две основные категории: (1) интерпретаторы и (2) компиляторы.

Интерпретатор (interpreter) читает исходный код программы, написанный программистом, анализирует код, и выполняет инструкции на лету (on-the-fly).

Python является интерпретатором, мы можем написать строку на Python, и он немедленно ее обработает и выдаст результат.

Некоторые строки говорят Python о том, что вы хотите их запомнить в некоторой переменной (variable). Нам необходимо выбрать имя для переменной, позже мы можем воспользоваться этим именем, чтобы получить значение, которое сохранили ранее.

```
>>> x = 6
>>> print x
6
>>> y = x * 7
>>> print y
42
>>>
```

В этом примере, мы просим Python запомнить значение 6 и используем метку x, по которой сможем восстановить значение позже. Мы убедились, что Python фактически вспоминает значение, используя print. Затем мы просим Python взять x, умножить его на 7 и полученный

результат сохранить в переменной y. После этого мы просим Python вывести на экран значение, хранящееся в переменной y.

Хотя мы вводим команды в Python на разных строках, они рассматриваются в качестве упорядоченной последовательности инструкций, мы можем обратиться к данным, созданным ранее.

Сущность интерпретатора заключается в возможности интерактивного общения, как было показано выше. Компилятор нуждается в передаче всей программы в файл, после этого запускается процесс трансляции с языка высокого уровня в машинный язык и только после этого компилятор помещает результат машинного языка в файл, который позже будет исполняться.

Если у вас ОС Windows, то часто программа, содержащая исполняемые машинные инструкции имеет расширение ".exe" или ".dll", "исполняемый файл" (executable) или "динамически загружаемая библиотека" (dynamically loadable library") соответственно. В Linux и Macintosh нет расширения, который бы однозначно отмечал файл как исполняемый.

Если вы откроете исполняемый файл с текстовом редакторе, выглядит все совершенно ненормально и не читаемо:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\x
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^
....
```

Не просто читать и писать на машинном языке, поэтому хорошо, что у нас есть интерпретаторы и компиляторы, которые позволяют нам писать на языке высокого уровня, таком как Python.

На данный момент в нашем обсуждении компиляторов и интерпретаторов, вам должно быть стало интересно узнать об интерпретаторе Python. На каком языке он написан? Написан ли он на компилируемом языке? Когда мы запускаем Python, что на самом деле происходит?

Интерпретатор Python написан на языке программирования Си. Вы можете посмотреть исходные коды Python на сайте : ссылка: www.python.org. В ОС Windows исполняемый машинный код для Python, скорее всего, будет находиться в файле с именем:

```
C:\Python27\python.exe
```

1.7. Написание программ

Ввод команд в интерпретатор Python – отличная возможность для экспериментов с особенностями языка Python, но это не рекомендуется делать при решении более сложных задач.

Когда мы хотим написать программу, мы используем текстовый редактор для написания инструкций в файл, который называется скриптом (script). По договоренности, скрипты на Python имеют расширение .py.

1.8. Что такое программа?

Наиболее общее определение программы – последовательность инструкций на языке Python, которая создана, чтобы что-то делать. Наш простой скрипт hello.py – это программа. Программа может содержать инструкцию, состоящую из одной строки.

Предположим, что вы проводите исследование сообщений в социальной сети Facebook, вам интересны наиболее часто встречающиеся слова в последовательности сообщений. Вы можете вывести на экран поток сообщений и корпеть над поиском наиболее распространенных слов, но это займет много времени и может привести к ошибке. Вы могли бы быть достаточно умны, чтобы написать программу на Python, быстро решающую задачу, и освободить себе выходные для более интересных занятий.

Для примера посмотрим на следующий текст о клоунах и автомобилях. Посмотрите на текст и выделите наиболее часто встречающиеся слова и количество раз, сколько они встречаются.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

Представим, что вы выполняете это задание, просматривая миллионы строк текста. Честно говоря, это проще сделать с помощью программы на языке Python, которая подсчитывает количество слов в тексте.

Я написал программу, которая ищет наиболее часто повторяющиеся слова в тексте. Теперь вы можете воспользоваться этой программой, сэкономив себе немного времени:

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()

for word in words:
    counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
      bigword = word
      bigcount = count

print bigword, bigcount
```

Исходный текст программы доступен в архиве: ссылка: http://pycode.ru/files/python/words.zip

Вам даже не надо знать Python, чтобы воспользоваться этой программой. Всю необходимую информацию по внутренней работе программы вы получите в главе 10 данного пособия.

Это хороший пример того, как интерпретатор Python и язык программирования Python выступают посредниками между вам (конечными пользователями) и мной (разработчиком). Python является способом обмена полезными инструкциями (например, программами)

на общем языке, который доступен всем, кто установил Python на свои компьютеры. Таким образом, мы общаемся не с Python, а между собой посредствам Python.

1.9. Построение частей программ

В следующих нескольких главах, мы узнаем чуть больше о словаре, структуре фраз и отступов. Мы узнаем о мощных возможностях Python и как совместить эти возможности, чтобы создавать полезные программы.

Есть несколько важных шаблонов, которые используются при построении программы. Эти конструкции относятся не только к Python, они являются частью любого языка программирования, начиная от машинного языка и заканчивая языками высокого уровня.

- Входные данные (input): получение данных из внешнего мира. Это может быть чтение данных из файла или из других источников, например, микрофона или GPS. В наших первых программах, входные данные будут задаваться пользователем с клавиатуры.

- Выходные данные (output):отображение результатов работы программы на экране, сохранение их в файл или, возможно, воспроизведение в виде музыки или голоса.

- Последовательное исполнение (sequential execution):инструкции выполняются в том порядке, в котором они встречаются в скрипте.

- Условное выполнение (conditional execution):проверка определенных условий и выполнение/пропуск последовательности инструкций.

- Повторное выполнение (repeated execution):выполнять некоторый набор команд несколько раз, обычно с некоторыми изменениями.

- Повторное использование (reuse):один раз написать набор инструкций, присвоить ему имя и затем повторно использовать

этот набор во всей программе.

1.11. Словарь

Ошибка (bug): ошибка в программе.

Центральный процессор (central processing unit): сердце компьютера, он исполняет программы, которые мы пишем.

Компиляция (compile): преобразование программы, написанной на языке высокого уровня, в низкоуровневый язык и ее подготовка для последующего выполнения.

Язык высокого уровня (high-level language): язык программирования, подобный Python, который разрабатывался, чтобы быть понятым для чтения и написания человеком.

Интерактивный режим (interactive mode): способ использования интерпретатора Python в режиме ввода команд и выражений.

Интерпретация (interpret): построчное исполнение программы на языке высокого уровня.

Язык программирования низкого уровня (low-level language): язык программирования, который разрабатывался, чтобы быть простым для исполнения компьютером; он называется "машинным кодом" или "языком ассемблера".

Машинный код (machine code): низкоуровневый язык программирования для программного обеспечения, который напрямую выполняется CPU.

Оперативная память (main memory): хранит программы и данные, теряет всю информацию, когда прекращается питание компьютера.

Разбор (parse): исследование программы и анализ синтаксиса.

Переносимость (portability): свойство программы, которое позволяет ей выполняться на более, чем одном виде компьютера.

Оператор печати (print statement): инструкция, которая отображает значение на экране.

Решение задачи (problem solving): процесс формулировки задачи, поиска решения и реализации.

Программа (program): набор инструкций, предназначенных для обработки на компьютере.

Приглашение (prompt): когда программа отображает сообщение и ожидает ввода пользовательских данных.

Вторичная память (secondary memory): хранит программы, данные и сохраняет информацию после выключения компьютера из сети питания. Примеры вторичной памяти: диски, флеш-память.

Семантика (semantics): смысл программы.

Семантическая ошибка (semantic error): ошибка в программе, когда происходит то, что программист не планировал.

Исходный текст (source code): программа на языке высокого уровня (high-level language).

Переменные, выражения и инструкции (Variables, expressions and statements)

Видео

2.1. Значения и типы

Значение (value) – одно из базовых понятий, с которым работают программы, наподобие букв или чисел. Значения могут выглядеть по-разному: 1, 2 или ‘Hello, World!’.

Эти значения принадлежат различным типам (types): 2 – это целочисленное значение (integer), ‘Hello, World!’ – строковое (string), т.к. содержит "строку" букв. Вы (и интерпретатор) можете распознать строки, т.к. они заключаются в кавычки

Оператор print также работает с целыми числами:

```
>>> print 4
4
```

Если вы не уверены в типе значения, интерпретатор может предоставить вам подсказку:

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Не удивительно, что строки принадлежат к типу str, а целые значения - к типу int. Менее очевидно то, что числа с дробной частью принадлежат к типу float, такой формат называется с плавающей точкой (floating-point).

```
>>> type(2.2)
<type 'float'>
```

Как насчет ‘17’ и ‘2.2’? Они похожи на числа, но кавычки указывают на то, что это строки:

```
>>> type('17')
<type 'str'>
>>> type('2.2')
<type 'str'>
```

Они являются строками.

Если вы встречаете большое число, то может возникнуть соблазн использовать разделители между группами цифр:

```
>>> print 1,000,000
1 0 0
```

Python интерпретировал 1,000,000 как последовательность целых чисел, разделенных запятой, и вывел на экран отдельные числа.

Мы столкнулись с примером семантической ошибки: код выполняется без сообщений об ошибке, но делает не то, что мы задумывали.

2.2. Переменные

Одна из наиболее мощных особенностей языка программирования – это манипуляция переменными (variables). Переменная – это имя, которое ссылается на значение.

Оператор присваивания (assignment statement) создает новые переменные и присваивает им значения:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 2.1415926535897931
```

Этот пример демонстрирует три оператора присваивания. Сначала строка присваивается переменной с именем message; затем значение 17 – переменной n; третье присваивание – приближенное значение числа pi присваивается переменной pi.

Общий способ представления переменных на бумаге заключается в написании имени со стрелкой, направленной на значение этой переменной. Подобный вид схемы называется диаграммой состояний

(state diagram), потому что показывает состояние каждой из переменных. Следующая диаграмма показывает результат из предыдущего примера:

Для отображения на экране значения переменной можно воспользоваться инструкцией print:

```
>>> print n
17
>>> print pi
2.14159265359
```

Типы переменных – это типы значений, на которые они ссылаются.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

2.2. Имена переменных и ключевые слова

Программисты обычно используют для своих переменных понятные имена, которые раскрывают их назначение.

Имена переменных могут иметь произвольную длину. Они могут содержать буквы и цифры, но они должны начинаться с буквы. Можно использовать буквы в верхнем регистре, но лучше имена переменных начинать со строчной буквы (вы позже узнаете почему).

Символ подчеркивания (_) может встречаться в имени переменной. Он часто используется в именах с несколькими словами, такими как my_name или airspeed_of_unladen_swallow.

Если вы зададите для переменной некорректное имя, то получите синтаксическую ошибку:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones – неправильное имя, т.к. оно не начинается с буквы. more@ - неправильное имя, т.к. содержит некорректный символ @. Что не так с class?

Слово class является зарезервированным (или ключевым) словом (keywords), оно используется Python для распознавания структуры программы и не может быть использовано в качестве имени переменной.

В Python зарезервировано 31 слово:

```
and del from not while as elif global or with assert else if
pass yield break except import print class exec in raise
continue finally is return def for lambda try
```

В новой версии Python 2.0, exec больше не является зарезервированным словом, а nonlocal – является.

2.4. Операторы

Инструкции (statement) – это единица измерения кода, которую интерпретатор Python может выполнять. Мы уже встречались с двумя видами инструкций: print (вывод на экран) и присваивание.

Когда вы вводите инструкцию в интерактивном режиме, интерпретатор выполняет ее и отображает результат.

Скрипт обычно содержит последовательность инструкций. Если инструкций несколько, то результат появляется одновременно с выполнением инструкции.

Например, выполнение скрипта:

```
print 1
x = 2
print x
```

приводит к результату

```
1
2
```

Инструкция присваивания ничего не выводит на экран.

2.5. Операторы и операнды

Операторы (operators) – специальные символы, которые представляют вычисления, наподобие сложения и умножения. Значения, к которым применяется оператор, называются операндами (operands).

Операторы +, -, *, / и ** выполняют соответственно сложение, вычитание, умножение и возведение в степень:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
0
```

Операция деления может привести к неожиданным результатам:

```
>>> minute = 59
>>> minute/60
0
```

Значение переменной minute равно 59, операция деления 59 на 60 приведет к результату 0.98333, а не к нулю! Причиной подобного результата в Python является округление (floor division).

Подробнее об исторических предпосылках округления можно узнать в блоге автора Python:

ссылка: http://python-history.blogspot.com/2010/08/why-pythons-integer-divisionfloors.html

Если оба операнда целочисленные, то результат тоже будет целочисленным. В нашем примере дробная часть отбрасывается, и в результате получаем нуль.

Если какой-либо из операндов является числом с плавающей точкой, то результат тоже будет типа float:

```
>>> minute/60.0
0.9833333333333333
```

В новой версии Python 2.0, результат деления целочисленных значений будет числом с плавающей точкой.

2.6. Выражения

Выражение (expression) – это комбинация значений, переменных и операторов. Значение (или переменная) само по себе является выражением, поэтому все следующие выражения являются корректными (предполагается, что переменной x было присвоено значение):

```
17
x
x + 17
```

Если выражения вводятся в интерактивном режиме, то интерпретатор вычисляет (evaluates) их и отображает результат:

```
>>> 1 + 1
2
```

В скрипте подобное выполняться не будет. Вывод результата без использования print, работает только в интерактивном режиме Python!

2.7. Порядок операций

Если в выражении встречается больше, чем один оператор, то порядок вычислений зависит от правил старшинства (rules of precedence). Для математических операций, Python следует математическим соглашениям. Аббревиатура PEMDAS является простым способом для запоминания правил:

- Скобки (Parentheses) имеют наивысший приоритет и могут использоваться для принудительного определения порядка вычислений в выражении. Таким образом, результат выражения 2*(3-1) будет равен 4, (1+1)**(5-2) будет равен 8. Вы также можете использовать скобки для упрощения чтения выражений, например, (minute*100) / 60, если это не повлияет на результат.

- Возведение в степень (Exponentiation) имеет наибольший приоритет, так 2**1+1 равно 3, а не 4, и 3*1**3 равно 3, а не 27.

- Умножение и деление (Multiplication and Division) имеют одинаковый приоритет, который выше сложение и вычитания (Addition and Subtraction), которые также имеют одинаковый приоритет. Таким образом, 2*3-1 равно 5, а не 4, и 6+4/2 равно 8, а не 5.

- Операторы с одинаковым приоритетом вычисляются слева направо. Таким образом, 5-3-1 равно 1, а не 2.

Если вы сомневаетесь, то поставьте скобки.

2.8. Модульные операторы

Модульные операторы работают с целочисленными значениями и возвращают остаток от деления (yields the remainder) двух операндов. В Python модульный оператор представлен символом (%). Синтаксис у него следующий:

```
>>> quotient = 7 / 3
>>> print quotient
```

```
2
>>> remainder = 7 % 3
>>> print remainder
1
```

Модульный оператор может быть очень полезен, если вы хотите проверить делится ли x на y без остатка – если x % y равно 0.

Также, вы можете извлечь самую правую цифру или цифры из числа. Например, x % 10 вернет самую правую цифру числа x (по основанию 10).

2.9. Строковые операции

Оператор + работает со строками, но это не сложение в математическом смысле. Вместо этого, выполняется операция конкатенации (concatenation), которая объединяет строки одна за другой. Например,

```
>>> first = 10
>>> second = 15
>>> print first+second
25
>>> first = '100'
>>> second = '150'
>>> print first + second
100150
```

2.10. Ввод входных данных

Иногда необходимо вводить значение переменной, используя клавиатуру. Python предоставляет встроенную функцию raw_input, которая получает входные значения с клавиатуры. В новой версии Python 2.0 эта функция носит название input. Когда вызывается эта функция, программа останавливается и ожидает действий пользователя. Когда пользователь нажимает Return или Enter, программа продолжает выполнение и raw_input возвращает введенную пользователем строку.

```
>>> input = raw_input()
```

```
Some silly stuff
>>> print input
Some silly stuff
```

Перед тем, как получить от пользователя некоторые данные, будет хорошим тоном – вывести на экран информацию о том, что пользователю необходимо ввести. Вы можете передать эту информацию в качестве входного параметра функции raw_input:

```
>>> name = raw_input('What is your name?\n')
What is your name?
Chuck
>>> print name
Chuck
```

Последовательность \n в конце приглашения обозначает новую строку (newline), поэтому пользователь начинает ввод со следующей после приглашения строки.

Если вы ожидаете от пользователя ввода целочисленного значения, то можете воспользоваться функцией int для преобразования входной строки в число:

```
>>> prompt = 'What...is the airspeed velocity of an unladen
swallow?\n'
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

Не всегда преобразование типов проходит гладко:

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
```

```
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    int(speed)

ValueError: invalid literal for int() with base 10: 'What do you
mean, an African or a European swallow?'
```

Позже мы рассмотрим, как обрабатывать подобные ошибки.

2.11. Комментарии

Когда программы разрастаются то, их становится сложно читать. Чтобы упростить чтение отдельных частей программы в программу добавляются заметки, которые на человеческом языке рассказывают, что программа делает. Эти заметки называются комментариями и начинаются с символа #.

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

Комментарии можно оставлять в конце выражения:

```
percentage = (minute * 100) / 60 # percentage of an hour
```

Весь текст, начиная от # и до конца строки, игнорируется Python.

Комментарии призваны ответить на вопрос, почему? Они обычно содержат полезную информацию, которой НЕТ в коде.

2.14. Словарь

Присваивание (assignment): выражение, в котором некоторое значение присваивается некоторой переменной.

Конкатенация (concatenate): объединение двух операндов один за другим.

Комментарий (comment): информация в программе, которая

предназначена для программистов (или тех, кто читает исходный код) и не влияет на выполнение программы.

Выражение (expression): комбинация переменных, операторов и значений, представленных единственным результирующим значением (single result value).

Плавающая точка (floating-point): тип представления чисел с дробной частью.

Остаток от деления (floor division): операция, которая делит два числа и отсекает дробную часть.

Целое число (integer): тип представления целых чисел.

Ключевые слова (keyword): зарезервированные слова, которые используются компилятором для анализа программы; эти слова нельзя использовать в качестве имен переменных.

Модульный оператор (modulus operator): оператор, представленный символом (%), который работает с целыми числами и возвращает остаток от деления двух чисел.

Операнд (operand): одно из значений, которым оперирует оператор.

Оператор (operator): специальный символ, который представляет простые вычисления, подобные сложению, умножению или конкатенации строк.

Правила старшинства (rules of precedence): набор правил, которые определяют порядок вычисления выражений, содержащих несколько операторов и операндов.

Диаграмма состояний (state diagram): графическое представление множества переменных и связанных с ними значений.

Инструкция (statement): фрагмент кода, представляющий команду или действие. Например, инструкция присваивания или вывода на экран.

Строка (string): тип представления последовательности символов.

Тип (type): категории значений. Мы уже встречали целые (int), с плавающей точкой (float) и строковый (str).

Значение (value): один из базовых элементов данных, подобно числам или строкам, которыми манипулирует программа.

Переменная (variable): имя, связанное со значением.

Программа "Hello, World!"

Видео

Программа "Почасовая оплата"

Видео

Условное выполнение

Видео

5.1. Логические выражения

Логическими (boolean expression) называются выражения, которые могут принимать одно из двух значений – истина или ложь. В следующем примере используется оператор ==, который сравнивает два операнда и возвращает True, если они равны (equal) или в противном случае False:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True и False – специальные значения, которые принадлежат к типу bool, они не являются строками:

```
>>> type(True)
type 'bool'
>>> type(False)
type 'bool'
>>>
```

Оператор == является одним из операторов сравнения (comparison operators), другие операторы сравнения:

```
x != y # x не равны y
x > y # x больше y
x < y # x меньше y
x >= y # x больше или равно y
x <= y # x меньше или равно y
x is y # x такой же, как y
x is not y # x не такой же, как y
```

Все эти операции, возможно, вам знакомы, но в Python их смысл несколько отличается от принятого в математике. Распространенная ошибка заключается в использовании одиночного символа =, вместо

двойного ==. Запомните, что = является оператором присваивания, а == является оператором сравнения. В Python нет подобного =< или =>.

5.2. Логические операторы

Существуют три логических оператора (logical operators): and, or и not. Смысл этих операторов схож с их смыслом в английском языке:

```
x > 0 and x < 10
```

это истинно в случае, если x больше 0 и меньше 10.

```
n%2 == 0 or n%3 == 0
```

это истинно, если одно из выражений является истинным.

В заключение, оператор not отрицает логическое выражение, так

```
not (x > y)
```

истинно, если x > y является ложью, т.е. x меньше или равно y.

Строго говоря, операнды логических операторов должны быть логическими выражениями, но Python не очень требователен к этому. Любое ненулевое число интерпретируется им как "истинное".

```
>>> 17 and True
True
```

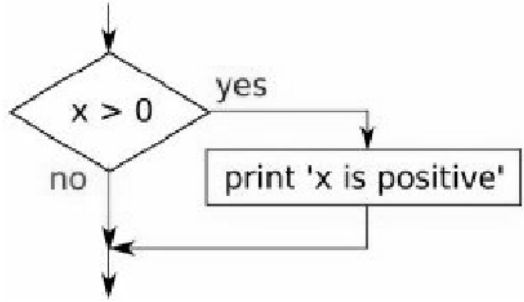
5.3. Условное исполнение

Для того чтобы писать полезные программы, почти всегда необходимо проверять условия и изменять поведение программы. Условные инструкции (conditional statements) предоставляют нам такую возможность. Простейшей формой является инструкция if:

```
if x > 0 :
print 'x is positive'
```

Логическое выражение после инструкции if называется условием

(condition). Далее следует символ двоеточия (:) и строка (строки) с отступом.



Если логическое условие истинно, тогда управление получает выражение, записанное с отступами, иначе – выражение пропускается.

Инструкция if имеет схожую структуру с определением функции или цикла for. Инструкция содержит строку заголовка, которая заканчивается символом двоеточия (:), за которым следует блок с отступом. Подобные инструкции называются составными (compound statements), т.к. содержат больше одной строки.

Не существует ограничения на число инструкций, которые могут встречаться в теле if, но хотя бы одна инструкция там должна быть. Иногда полезно иметь тело if без инструкций (обычно оставляют место для кода, который еще не написан). В этом случае можно воспользоваться инструкцией pass, которая ничего не делает.

```
if x < 0 :
pass # необходимо обработать отрицательные значения!
```

Если вы вводите инструкцию if в интерпретаторе Python, то приглашение сменится с трех угловых скобок на три точки (в версии 2.7 наблюдается отступ вместо точек), которые сигнализируют о том, что вы находитесь в блоке инструкций:

```
>>> x = 3
>>> if x < 10:
        print 'Small'
```

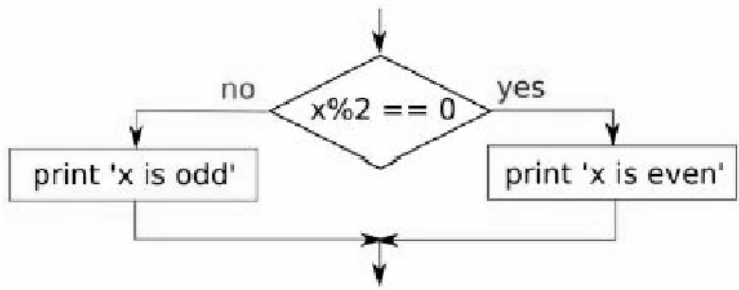
```
Small
>>>
```

5.4. Альтернативное исполнение

Второй формой инструкции if является альтернативное исполнение (alternative execution), в котором существуют два направления выполнения и условие определяет, какое из них выполнится. Синтаксис выглядит следующим образом:

```
if x%2 == 0 :
        print 'x is even'
else :
        print 'x is odd'
```

Если остаток от деления x на 2 равен 0, то x-четное, и программа выводит сообщение об этом. Если условие ложно, то выполняется второй набор инструкций.



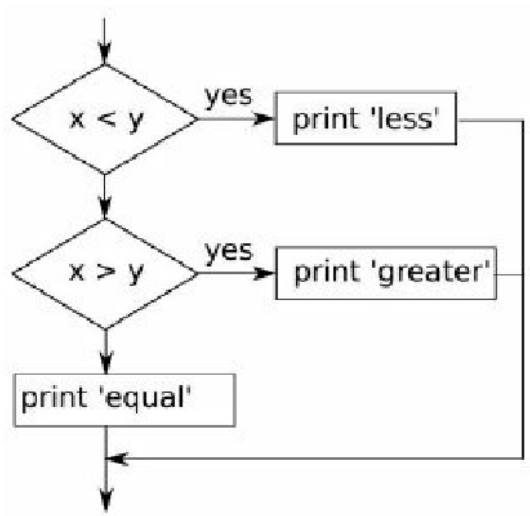
Так как условие может быть либо истинным, либо ложным, тогда точно выполнится один из вариантов. Варианты называются ветвями (branches), потому что они являются ответвлениями в потоке исполнения.

5.5. Последовательность условий

Иногда имеется больше двух вариантов выполнения, тогда нам необходимо больше двух ветвей. В этом случае, можно воспользоваться сцепленными условиями (chained conditional):

```
if x < y:
    print 'x is less than y'
elif x > y:
    print 'x is greater than y'
else:
    print 'x and y are equal'
```

elif является аббревиатурой от "else if". Снова будет исполнена точно одна ветвь.



Не существует ограничения на количество инструкций elif. Если встречается оператор else, то он должен быть в конце, но может не быть ни одного.

```
if choice == 'a':
print 'Bad guess'
elif choice == 'b':
print 'Good guess'
elif choice == 'c':
print 'Close, but not correct'
```

Каждое условие проверяется в порядке расположения. Если первое условие ложно, то проверяется следующее и т.д. Если одно из условий истинно, то выполняется соответствующая ветка, и инструкция завершается. Даже если верно более чем одно условие, все равно

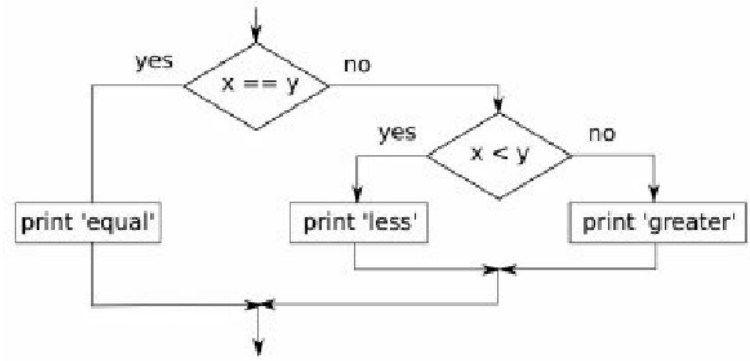
выполняется только первая истинная ветка.

5.5. Вложенные условия

Одно условие может быть вложено в другое. Мы можем записать пример трихотомии (trichotomy):

```
if x == y:
print 'x and y are equal'
else:
if x < y:
print 'x is less than y'
else:
print 'x is greater than y'
```

Внешнее условие содержит две ветки. Первая ветка содержит простую инструкцию. Вторая ветка содержит еще одну инструкцию if, которая имеет две ветки. Эти две ветки являются простыми инструкциями, хотя они также могут содержать инструкции.



Хотя отступ инструкций делает структуру более очевидной, вложенные условия (nested conditionals) усложняют чтение исходного кода. Избегайте их по возможности.

Логические операторы часто позволяют упростить вложенные условные инструкции. Например, мы можем записать следующий код, используя одно условие:

```
if 0 < x:
```

```
    if x < 10:
        print 'x is a positive single-digit number.'
```

Инструкция print выполнится только, если мы зададим ее после обоих условий, также мы получим похожий эффект, используя оператор and:

```
if 0 < x and x < 10:
    print 'x is a positive single-digit number.'
```

5.7. Перехват исключений с использованием try и except

Ранее мы рассматривали код, где использовались функции raw_input и int. Мы также видели их ненадежное исполнение:

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
>>>
```

Когда мы запускали эти инструкции в интерпретаторе Python, то получали строку приглашения, затем набирали наш текст.

Пример простой программы, которая перевод температуру по Фаренгейту в температуру по Цельсию:

```
inp = raw_input('Enter Fahrenheit Temperature:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

Если мы запустим этот код и введем некорректные данные, то получим недружелюбное сообщение об ошибке:

```
Enter Fahrenheit Temperature:72
22.2222222222
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
File "fahren.py", line 2, in <module>
```

```
  fahr = float(inp)
  ValueError: invalid literal for float(): fred
```

Существует структура условного выполнения, встроенная в Python, которая обрабатывает эти типы ожидаемых и неожиданных ошибок, она называется "try / except". Идея try и except заключается в следующем: вы знаете, что некоторая последовательность инструкций может иметь проблемы, и вы хотите добавить некоторые инструкции, которые бы выполнялись в случае возникновения ошибки. Эти дополнительные инструкции (блок except) игнорируются, если ошибок не произошло.

Вы можете представить try и except как страховой полис для последовательности инструкций.

Перепишем нашу программу о температуре следующим образом:

```
inp = raw_input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print cel
except:
    print 'Please enter a number'
```

Python начинает выполнение с последовательности инструкций в блоке try. Если все выполняется без ошибок, то блок except пропускается. Если произошло исключение (exception) в блоке try, то Python покидает блок try и выполняет последовательность инструкций внутри блока except.

```
Enter Fahrenheit Temperature:72
22.2222222222


Enter Fahrenheit Temperature:fred
Please enter a number
```

Обработка исключения с помощью инструкции try называется перехватом (catching) исключения. В рассмотренном примере, блок except выводит на экран сообщение об ошибке. В общем, перехват исключения предоставляет вам шанс решить проблему, попытаться

заново или хотя бы красиво завершить работу программы.

5.10. Словарь

Тело (body): последовательность инструкций в составной инструкции.

Логическое выражение (boolean expression): выражение, значением которого может быть либо True, либо False.

Ветка (branch): одна из возможных последовательностей инструкций в условной инструкции.

Цепочное условие (chained conditional): условная инструкция с несколькими возможными ветками.

Оператор сравнения (comparison operator): один из операторов, который сравнивает операнды: ==, !=, >, <, >= и <=.

Условная инструкция (conditional statement): инструкция, которая контролирует поток выполнения в зависимости от некоторого условия.

Условие (condition): логическое выражение в условной инструкции, которое определяет, какая ветка выполнится.

Составная инструкция (compound statement): инструкция, которая содержит заголовок и тело. Заголовок заканчивается (:). Тело смещается относительно заголовка.

Логический оператор (logical operator): один из операторов, которые объединяют логические выражения: and, or и not.

Вложенное условие (nested conditional): условная инструкция, которая встречается в одной из веток другой условной инструкции.

Программа "Почасовая оплата труда с учетом переработок"

Видео

Усовершенствование программы "Почасовая оплата труда с учетом переработок"

Видео

Функции

Видео

8.1. Вызов функции

В контексте программирования, функцией (function) называется последовательность инструкций, которая выполняет вычисления. Когда вы задаете функцию, то задаете имя и последовательность инструкций. Позже вы сможете вызвать функцию, обратившись к ней по имени. Мы уже встречали примеры вызова функции (function call):

```
>>> type(32)
<type 'int'>
```

Функция носит имя `type`. Выражение в скобках называется аргументом (argument) функции. Аргумент – это значение или переменная, которую мы передаем в функцию в качестве входного параметра. Результатом функции type является тип аргумента.

Проще говоря, функция "берет" аргумент и "возвращает" результат. Результат называется возвращаемым значением (return value).

8.2. Встроенные функции

Python предоставляет несколько важных (built-in) встроенных функций, которые мы можем использовать. Создатели Python написали множество функций, которые решают часто встречающиеся проблемы и включил эти функции в состав Python.

Функции max и min возвращают соответственно наибольшее и наименьшее значения в списке:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
```

Функция `max` говорит нам о "наибольшем символе" в строке (который функция нам вернула как "w"), функция `min` показывает наименьший символ, которым является пробел.

Другая очень распространенная встроенная функция - `len`, которая говорит нам, сколько элементов содержится в аргументе. Если аргументом функции len является строка, то она возвращает количество символов в строке.

```
>>> len('Hello world')
11
>>>
```

Эти функции не исчерпываются просмотром строк, они могут оперировать любым множеством значений, как мы увидим в следующих главах.

Вы должны обращаться с именами встроенных функций, как с зарезервированными словами (т.е. не используйте max в качестве имени переменной).

8.3. Функции, преобразующие типы

Python также предоставляет встроенные функции, которые преобразуют значения из одного типа в другой. Функция `int` берет любое значение и преобразует его в целое число, если это возможно:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

Функция `int` может преобразовать число с плавающей точкой в целое, но это не округление, а отсечение дробной части:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

Функция `float` преобразует целое и строковое в число с плавающей точкой:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

В завершении, функция `str` преобразует входные аргументы в строки:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

8.4. Случайные числа

Получая одинаковые входные значения, большинство программ каждый раз генерируют одни и те же выходные значения, такие программы можно назвать детерминированными (deterministic). Детерминизм обычно является полезной вещью, но для некоторых приложений необходима непредсказуемость в поведении. Хорошим примером таких приложений являются игры.

Создание действительно недетерминированных программ – дело не простое, но есть пути создания программ, которые похожи на недетерминированные. Одним из таких способов являются алгоритмы (algorithms) генерации псевдослучайных (pseudorandom) чисел. Псевдослучайные числа не настоящие случайные, т.к. они генерируются на основании детерминированных вычислений, но с виду их не отличить от случайных.

Модуль `random` предоставляет функции, которые генерируют псевдослучайные числа (которые далее мы будем называть "случайными").

Функция `random` возвращает случайное число с плавающей точкой в интервале от 0.0 до 1.0 (включая 0.0 и не включая 1.0). Каждый раз, когда вызывается `random`, вы получаете следующее число в длинной

последовательности. Для примера рассмотрим цикл:

```
import random
for i in range(10):
    x = random.random()
    print x
```

Результатом работы программы будет список из следующих 10 случайных чисел в интервале от 0.0 до 1.0.

```
0.301927091705
0.513787075867
0.319470430881
0.285145917252
0.839069045123
0.322027080731
0.550722110248
0.366591677812
0.396981483964
0.838116437404
```

Функция `random` является одной из функций, которая работает со случайными числами. Функция `randint` принимает параметры `low` и `high`, и возвращает целочисленное значение в интервале от `low` до `high` (включая оба).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Для выбора случайного элемента из последовательности, можно воспользоваться функцией choice:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Модуль `random` также включает функции, которые генерируют значения от непрерывных распределений, включая Гаусса, экспоненциального, гамма и некоторых других.

8.5. Математические функции

В Python содержится математический модуль (module), который предоставляет большинство популярных математических функций. Перед тем, как его использовать, нам необходимо его импортировать:

```
>>> import math
```

Эта инструкция создает модульный объект (module object), который называется math. Если вы выведите на экран модульный объект, то получите некоторую информацию о нем:

```
>>> print math
<module 'math' (built-in)>
```

Модульный объект содержит функции и переменные, определенные в объекте. Для получения доступа к одной из этих функций, вам необходимо задать имя модуля и имя функции, разделенные точкой (period). Этот формат называется точечной нотацией (dot notation).

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

В первом примере вычисляется логарифм по основанию 10 отношения "сигнал-шум". Модуль math также предоставляет функцию `log`, которая вычисляет логарифмы по основанию `e`.

Во втором примере вычисляется синус `radians`. Имя переменной подсказывает, что `sin` и другие тригонометрические функции (cos, tg и т.д.) принимают аргументы в радианах. Для перевода градусов в радианы, разделим на 360 и умножим на `2*pi`:

```
>>> degrees = 45
```

```
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

Выражение `math.pi` получает значение переменной `pi` из модуля `math`. Значением этой переменной является приближенное с точностью до 15 знаков.

8.6. Добавление новых функций

Ранее мы использовали встроенные в Python функции. Теперь мы рассмотрим, как добавлять новые функции. Определение функции (function definition) задает имя новой функции и последовательность инструкций, которые выполняются, когда функция вызывается. Как только мы определили функцию, мы можем многократно ее использовать.

Например,

```
def print_lyrics():
print "I'm a lumberjack, and I'm okay."
print 'I sleep all night and I work all day.'
```

`def` – это ключевое слово, которое показывает, что далее следует определение функции. Имя функции - `print_lyrics`. Правила наименования функций таки же, как для переменных: возможны буквы, числа и некоторые знаки пунктуации, но первая буква не может быть числом. Вы не можете использовать ключевые (зарезервированные) слова в качестве имен функций. Имена функций и переменных не должны совпадать.

Пустые скобки после имени указывают на то, что функция не принимает аргументов. Позже мы рассмотрим функции, которые принимают входные аргументы.

Первая строка определения функции называется заголовком (header), оставшаяся часть – телом (body) функции. Заголовок заканчивается двоеточием, тело функции имеет отступ. По договоренности, отступ всегда является четырьмя пробелами. Тело функции может содержать

любое количество инструкций.

Строки, которые мы выводим на экран, заключены в двойные кавычки. Одиночные и двойные кавычки взаимозаменяемы, большинство людей используют одиночные кавычки, за исключением тех случаев, когда одиночные кавычки встречаются внутри строки.

Если вы будете набирать функцию в интерактивном режиме, то Python для тела функции сделает отступы, а в конце необходимо будет ввести пустую строку.

Определение функции создает переменную с таким же именем.

```
>>> print print_lyrics
<function print_lyrics at 0xb7e99e9c>
>>> print type(print_lyrics)
<type 'function'>
```

Значением print_lyrics является функциональный объект (function object), который имеет тип 'function'.

Синтаксис вызова новой функции похож на вызов встроенной функции:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Однажды определив функцию, вы можете использовать ее внутри других функций. Для примера повторим строку-припев, воспользовавшись новой функцией:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

Затем вызовем repeat_lyrics:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
```

```
I sleep all night and I work all day.
```

8.7. Определение и использование

Если собрать вместе весь код из предыдущего параграфа, то получим следующее:

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
repeat_lyrics()
```

Эта программа содержит два определения функций: `print_lyrics` и `repeat_lyrics`. Определения функций получают управление подобно другим инструкциям, результатом является создание функционального объект. Инструкции внутри функции не получат управления, пока функция не будет вызвана.

Как и следовало ожидать, вы должны предварительно создать функцию, прежде чем сможете ее выполнить. Иными словами, определение функции должен быть выполнено перед ее первым запуском.

8.8. Поток исполнения

Для того чтобы убедиться, что функция определяется до ее первого использования, вы должны знать порядок, в котором выполняются инструкции, он называется потоком исполнения (flow of execution).

Исполнение обычно начинается с первой инструкции программы. Инструкции выполняются по одной сверху вниз.

Определение функций не изменяют порядок исполнения программы, но запомните, что инструкции внутри функции не исполняются до вызова функции.

Вызов функции подобен обходному пути в потоке исполнения. Вместо того чтобы перейти к следующей инструкции, поток переходит в тело функции, выполняет все инструкции внутри тела, и затем возвращается обратно в то место, которое он покинул в момент вызова функции.

Это звучит достаточно просто, пока вы не вспомните, что одна функция может вызывать другую. В то время как в середине одной функции, программа может выполнять инструкции из другой функции. Но во время выполнения этой новой функции, программа может выполнять еще одну функцию!

К счастью, Python следит, где находится, так что всякий раз, когда функция завершается, программа переходит обратно, в место вызова функции. Программа завершится, когда дойдет до конца программы.

В чем мораль этой неприглядной истории? Когда вы читаете программу, вам не всегда хочется читать сверху вниз. Иногда это имеет смысл, если вы следуете за потоком исполнения.

8.8. Параметры и аргументы

Некоторые из встроенных функций требуют передачи входных аргументов. Например, когда вы вызываете `math.sin`, вы передаете число в качестве входного аргумента. Некоторые функции принимают больше одного аргумента, например, в `math.pow` передается два аргумента: основание и степень.

Внутри функции, аргументы, присвоенные переменным, называются параметрами (parameters). Пример определения функции, которой передается аргумент:

```
def print_twice(bruce):
    print bruce
    print bruce
```

Эта функция присваивает аргумент параметру с именем `bruce`. Когда вызывается функция, она дважды выводит на экран значение параметра (что бы это ни было).

Эта функция работает с любыми значениями, которые могут быть выведены на экран.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

Для построения функций, определяемых пользователем, используются те же правила, что и для встроенных функций, поэтому мы можем использовать любое выражение в качестве аргумента для `print_twice`:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Аргумент вычисляется перед вызовом функции, поэтому в примере выражения 'Spam '*4 и math.cos(math.pi) вычисляются только один раз.

Вы можете использовать переменные в качестве аргументов:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Имя переменной, которое мы передаем в качестве аргумента (michael), не имеет ничего общего с параметром (bruce). Не важно, какое значение было передано в вызывающую функцию; здесь, в print_twice, мы называем все bruce.

8.10. Плодотворные (fruitful) функции и void-функции

Некоторые из функций, которые мы используем, такие, как математические функции, возвращают результаты (приносят плоды); за неимением лучшего названия, я называю их плодотворными функциями (fruitful functions). Другие функции, подобные `print_twice`, выполняют некоторые действия, но не возвращают значение. Они называются void-функциями.

Когда вы вызываете плодотворную функцию, вы почти всегда хотите сделать что-то с результатом, например, вы можете присвоить его переменной или использовать его как часть выражения:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Когда вы вызываете функцию в интерактивном режиме, Python отображает результат:

```
>>> math.sqrt(5)
2.2360679774997898
```

Но если вы вызываете плодотворную функцию в скрипте и не сохраняете результат выполнения функции в переменной, то возвращаемое значение растворится в тумане!

```
math.sqrt(5)
```

Этот скрипт вычисляет квадратный корень из 5, но так как он не сохраняет результат в переменной и не отображает результат, то это не очень полезно.

Void-функции могут отображать что-то на экране или производить другие действия, но они не возвращают значение. Если вы попытаетесь присвоить результат выполнения такой функции переменной, то получите специальное значение, называемое None.

```
>>> result = print_twice('Bing')
Bing
Bing
```

```
>>> print result
None
```

Значение None – это не тоже самое, что строка ‘None’. Это специальное значение, которое имеет собственный тип:

```
>>> print type(None)
<type 'NoneType'>
```

Для возврата результата из функции, мы используем инструкцию return в функции. Для примера, мы можем создать простую функцию с именем addtwo, которая складывает два числа и возвращает результат.

```
def addtwo(a, b):
    added = a + b
    return added
x = addtwo(3, 5)
print x
```

Когда скрипт выполнится, инструкция print напечатает "8". Внутри функции параметры a и b, в момент выполнения, будут содержать значения 3 и 5 соответственно. Функция подсчитывает сумму двух чисел и помещает ее в локальную переменную `added`. Затем с помощью инструкции return результат вычисления возвращается в вызываемый код, как результат выполнения функции. Далее результат присваивается переменной `x` и выводится на экран.

8.11. Зачем нужны функции?

Есть несколько причин, на основании которых программу стоит разбивать на функции.

- Создание новой функции предоставляет вам возможность присвоить имя группе инструкций. Это позволит упростить чтение, понимание и отладку программы.
- Функции позволяют сократить код программы, благодаря ликвидации повторяющихся участков кода. Позже, вы сможете вносить коррективы только в одном месте.

- Разбиение длинной программы на функции позволяет одновременной отлаживать отдельные части, а затем собрать их в единое целое.

- Хорошо спроектированная функция может использоваться во множестве программ. Однажды написав и отладив функцию, вы можете использовать ее множество раз.

8.13. Словарь

Алгоритм (algorithm): обобщенный подход к решению категории проблем.

Аргумент (argument): значение, передаваемое функции, в момент ее вызова. Это значение присваивается соответствующему параметру в функции.

Тело функции (body): последовательность инструкций внутри определения функции.

Детерминистический (deterministic): относится к программе, которая выполняет одинаковые действия при одинаковых входных значениях.

Точечная нотация (dot notation): синтаксис для вызова функции из другого модуля.

Поток исполнения (flow of execution): порядок, в котором выполняются инструкции во время работы программы.

Плодотворная функция (fruitful function): функция, которая возвращает значение.

Функция (function): это именованная последовательность инструкций, которая выполняет некоторые полезные действия. Функции могут иметь или не иметь аргументы, могут возвращать или не возвращать результат.

Вызов функции (function call): инструкция, которая выполняет функцию. Она содержит имя функции и список аргументов.

Определение функции (function definition): инструкция, которая создает новую функцию; задает имя функции, параметры и инструкции для выполнения.

Функциональный объект (function object): значение, созданное определением функции. Имя функции является переменной, которая ссылается на функциональный объект.

Заголовок функции (header): первая строка в определении функции.

Инструкция импорта (import statement): инструкция, которая читает файл модуля и создает модульный объект.

Модульный объект (module object): значение, которое создает инструкция импорта, для предоставления доступа к данным и коду, определенном в модуле.

Параметр (parameter): имя, используемое внутри функции, которое ссылается на передающееся в качестве аргумента значение.

Псевдослучайный (pseudorandom): относится к последовательности чисел, которые похожи на случайные, но генерируются детерминированной программой.

Возвращаемое значение (return value): результат выполнения функции.

Void-функция (void function): функция, которая не возвращает результирующего значения.

Создаем первую функцию

Видео

Итерации

Видео

10.1. Обновление переменной

Общим шаблоном в инструкциях присваивания является инструкция присваивания, которая обновляет переменную, где новое значение переменной зависит от старого:

```
x = x+1
```

Это означает "получить текущее значение x, прибавить к нему 1 и затем обновить x, присвоив ему новое значение".

Если вы попытаетесь обновить переменную, которая не существует, то получите ошибку, т.к. Python вычисляет правую сторону раньше ее присваивания переменной x:

```
>>> x = x+1

NameError: name 'x' is not defined
```

Перед тем как обновить переменную, вам необходимо ее инициализировать (initialize), обычно с помощью простого присваивания:

```
>>> x = 0

>>> x = x+1
```

Обновление переменной, путем прибавления к ней 1, называется инкрементом (increment), вычитание 1, называется декрементом (decrement).

10.2. Инструкция while

Компьютеры часто используются для автоматизации повторяющихся задач. Повторение сходных или простых задач без ошибок – это то, что

компьютер делает лучше, чем человек. Поскольку итерации распространены, Python предоставляет несколько языковых особенностей для их создания.

Одной из форм итераций в Python является инструкция `while`. Далее представлена простая программа, которая производит обратный отсчет от 5 и затем говорит "Blastoff!".

```
n = 5
while n > 0:
        print n
        n = n-1
print 'Blastoff!'
```

Вы также можете читать инструкцию `while`, как если бы это был английский язык. Это означает, "до тех пор, пока n больше 0, выводить на экран значение n и уменьшать n на 1. Когда достигнем 0, покинуть инструкцию `while` и вывести на экран слово Blastoff!"

Более формально, поток выполнения для инструкции `while` имеет следующий вид:

1. Вычисление условия, получаем True или False.

2. Если условие ложно, то выходим из инструкции while и продолжаем выполнение со следующей инструкции.

3. Если условие истинно, то выполняем тело и затем возвращаемся на шаг 1.

Этот тип потока называется циклом (loop), т.к. третий шаг цикла возвращается на начало. Выполнение тела цикла называется итерацией (iteration). Для приведенного выше цикла, мы бы сказали, что "прошло 5 итераций", т.е. тело цикла было выполнено 5 раз.

Тело цикла должно изменять одно или более переменных, что в конечном итоге приведет к ложности условия и завершению цикла. Переменные, которые изменяются каждый раз при выполнении цикла и контролируют завершение цикла, называются итерационными

переменными (iteration variable). Если итерационная переменная в цикле отсутствует, то такой цикл будет бесконечным (infinite loop).

10.3. Бесконечные циклы

Бесконечным источником развлечения для программистов является высказывание на руководстве по шампуню: "Намылить, промыть, повторить". Это бесконечный цикл, поскольку в нем отсутствует итерационная переменная, которая указывает на количество выполнений в цикле.

В случае с обратным отсчетом, мы можем удостовериться, что цикл завершится, т.к. мы знаем, что значение n конечно, мы можем увидеть, что значение n каждый раз уменьшается и в конечном итоге станет равным нулю.

10.4. "Бесконечные циклы" и break

Иногда вы не знаете, когда завершить цикл, пока не достигните некоторого значения в теле. В этом случае, вы можете специально написать бесконечный цикл и затем использовать инструкцию `break`, чтобы из него выйти.

Это явно бесконечный цикл, т.к. логическое выражение в инструкции `while` содержит True:

```
n = 10
while True:
        print n,
         n = n - 1
print 'Done!'
```

Если вы совершите ошибку и запустите этот код, то быстро научитесь, как останавливать процесс Python в вашей системе или найдете кнопку выключения питания на вашем компьютере (что делать не желательно). Эта программа работает постоянно или пока не разрядится батарея, т.к. логическое выражение всегда будет истинным.

Мы можем добавить в тело цикла код с `break`, который позволит выйти из цикла при наступлении заданного условия.

Например, предположим, что мы принимаем входные значения с клавиатуры, пока пользователь не наберет "done". Можем записать это следующим образом:

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line
print 'Done!'
```

Условие цикла является `True`, т.е. всегда истинным. Оно будет выполняться, пока на достигнет инструкции прерывания.

10.5. Завершение итерации с помощью continue

Иногда вы находитесь в итерации цикла и хотите завершить текущую итерацию и сразу перейти к следующей итерации. В этом случае можно воспользоваться инструкцией `continue`, которая пропускает текущую итерацию и переходит к следующей без завершения тела цикла.

Далее представлен пример цикла, который копирует данные, поступающие на его вход до тех пор, пока не будет введено "done". Если строка начинается с символа `#`, то она не будет выводиться на печать (похоже на комментарии в Python).

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line
print 'Done!'
```

Все строки, которые вводятся с клавиатуры, будут выведены на экран,

кроме тех, которые начинаются с символа #.

10.6. Определение циклов с помощью for

Иногда мы хотим пройти в цикле через множество (set) вещей, таких как список слов, строки в файле или список чисел. Когда у нас есть подобный список, мы можем построить определенный (definite) цикл с использованием инструкции `for`. Мы вызывали инструкцию `while` в неопределенных (indefinite) циклах, т.к. циклы работали до того момента, пока условие не станет `False`. Цикл `for` проходит через известное множество элементов, т.е. столько раз, сколько элементов содержится во множестве.

Синтаксис цикла `for` похож на цикл `while`, здесь есть инструкция for и тело цикла:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
      print 'Happy New Year:', friend
print 'Done!'
```

Переменная `friends` является списком (в следующих главах вы об этом узнаете подробнее) из трех строк, цикл for проходит через список и выполняет тело цикла для каждой из трех строк:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Посмотрите на цикл `for`, здесь `for` и `in` являются зарезервированными словами, friend и friends являются переменными.

```
for friend in friends:
      print ’Happy New Year’, friend
```

В частности, `friend` является итерационной переменной для цикла `for`.

Переменная `friend` изменяется для каждой итерации цикла и контролирует, когда цикл `for` завершится.

10.7. Шаблоны цикла

Часто мы используем циклы `for` и `while` для того, чтобы протий в цикле через элементы списка или содержимое файла в поисках наибольшего или наименьшего значения.

Эти циклы обычно строятся по следующей схеме:

- Инициализация одной или более переменных до начала выполнения цикла.
- Выполнение некоторых вычислений для каждого элемента в теле цикла, возможно, изменение переменных в теле цикла.
- Просмотр результирующих переменных, когда цикл завершился.

Воспользуемся списком чисел для демонстрации подходов и создания распространенных шаблонов.

10.7.1. Циклы подсчета

Например, для подсчета числа элементов в списке, мы можем написать следующий цикл `for`:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print 'Count: ', count
```

Мы установили начальное значение для переменной `count` равным нулю перед тем, как начать выполнение цикла. Наша итерационная переменная называется `itervar`.

В теле цикла мы добавляем 1 к текущему значению `count` для каждого

значения в списке. К моменту завершения цикла, значение переменной count будет содержать количество элементов.

Другой простой цикл подсчитывает сумму набора чисел:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print 'Total: ', total
```

Переменная `total` накапливает сумму всех чисел, иногда ее называют сумматором (accumulator).

На практике для подсчета числа элементов используют встроенную функцию `len()`, а для вычисления суммы элементов – `sum()`.

10.7.2. Циклы максимума и минимума

Для нахождения наибольшего значения в списке или последовательности, создадим следующий цикл:

```
largest = None
print 'Before:', largest
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print 'Loop:', itervar, largest
print 'Largest:', largest
```

Результат работы программы будет иметь вид:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

Переменная `largest` содержит наибольшее значение, которое существует на данный момент. Перед запуском цикла мы устанавливаем `largest` в `None`.

`None` – это специальная константа, которая может храниться в переменной и маркировать ее как "пустую".

Перед началом цикла, наибольшим значением является `None`. На первой итерации цикла значение 3 больше `None`, мы устанавливаем `largest` равным 3. На последней итерации в `largest` содержится наибольшее значение.

Код для вычисления наименьшего значения:

```
smallest = None
print 'Before:', smallest
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print 'Loop:', itervar, smallest
print 'Smallest:', smallest
```

Снова smallest содержит наименьшее значение.

Далее следует простая версия встроенной в Python функции min():

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
return smallest
```

В этой функции меньше кода, мы удалили все инструкции print.

10.9. Словарь

Сумматор (accumulator): переменная в цикле, которая используется для накопления суммарного результата.

Счетчик (counter): переменная, используемая в цикле для подсчета количества встречаемости какого-либо события. Мы инициализируем счетчик нулевым значением, затем инкрементируем его при наступлении какого-либо события.

Декремент (decrement): уменьшение значения (обычно на единицу).

Инициализация (initialize): присваивание начального значения переменной, которая в дальнейшем будет инкрементироваться (увеличиваться, часто на единицу).

Бесконечный цикл (infinite loop): цикл, в котором условие завершения никогда не наступят или для которого нет условия завершения.

Итерация (iteration): повторное выполнение множества инструкций, используемое при рекурсивном вызове функции или цикла.

Вычисляем среднее значение

Видео

Строки

Видео

12.1. Строка - это последовательность

Строка - это последовательность символов. Вы можете получить доступ к одному символу с помощью оператора скобок:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

Вторая инструкция извлекает символ из позиции 1, хранящейся в переменной fruit и помещает его в переменную letter.

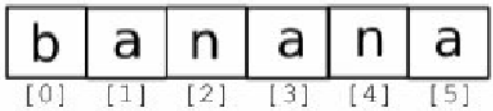
Выражение в скобках называется индексом (index). Индекс обозначает, какой символ в последовательности вы хотите получить:

```
>>> print letter
a
```

Для большинства людей, первая буква в 'banana' это `b`, а не `a`. Но в Python индекс - это смещение от начала строки, смещение для первого символа – нуль:

```
>>> letter = fruit[0]
>>> print letter
b
```

Таким образом, `b` - это нулевая буква 'banana', a - это первая буква и т.д.



Вы можете использовать любое выражение, включая переменные или операторы, в качестве индекса, но значение индекса должно быть целочисленным (`integer`):

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

12.2. Получение длины строки с использованием len

len - встроенная функция, которая возвращает число символов в строке:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

Чтобы получить последний символ в строке можно попробовать следующее:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

Причина IndexError в том, что не существует буквы с индексом 6 в строке 'banana', поэтому исправленный вариант:

```
>>> last = fruit[length-1]
>>> print last
a
```

Взамен можно использовать отрицательные индексы, которые считаются с конца. Выражение `fruit[-1]` выдаст последний символ, `fruit[-2]` - второй с конца строки и т.д.

12.3. Обход через строку с помощью цикла

Множество вычислений включают посимвольную обработку строк. Часто они начинаются с начала строки, выбирают каждый символ по очереди, делают что-то с ним и продолжают до окончания строки. Этот шаблон обработки называется обход (traversal). Один из вариантов написания обхода с помощью цикла while:

```
>>> index = 0
>>> while index < len(fruit):
```

```
    letter = fruit[index]
    print letter
    index = index + 1
```

```
b
a
n
a
n
a
```

Этот цикл обходит строку и отображает каждый символ строки отдельно.

Условием цикла является `index < len(fruit)`, т.е. когда `index` становится равным длине строки, условие становится ложным (false) и тело цикла прекращает выполняться.

Другой способ написания обхода с помощью цикла for:

```
>>> for char in fruit:
print char
```

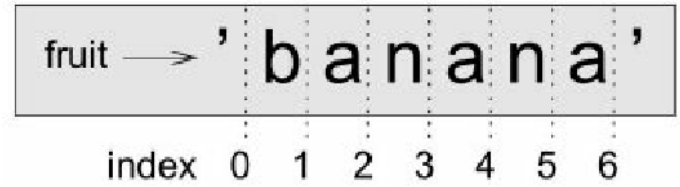
Каждый раз, в цикле следующий символ в строке присваивается переменной `char`. Цикл работает, пока не закончатся символы.

12.4. Срез строки

Часть строки называется срезом (slice). Выбор среза аналогичен выбору символа:

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:13]
Python
>>>
```

Оператор `[n:m]` возвращает часть строки от n-ого символа до m-ого символа, включая первый, но исключая последний. Это противоречивое поведение, но оно позволяет представить индексные указатели между символами, как на следующей диаграмме:



Если опустить первый индекс (перед двоеточием), то срез будет начинаться с начала строки. Если вы опустите второй индекс - срез завершится в конце строки:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

Если первый индекс больше или равен второму, то результатом будет пустая строка (`empty string`), представленная двумя кавычками:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

12.5. Строки являются неизменяемыми

Заманчиво использовать оператор `[ ]` с левой стороны при присвоении с намерением изменить символ в строке.

Например:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

Объект (object) в этом случае является строкой, элемент (item) - символ, который пытаемся изменить. На данный момент будем считать, что объект - некоторая вещь как переменная, но мы уточним это определение позже.

Элемент - одно из значений последовательности.

Причина ошибки в том, что строки являются неизменяемыми (immutable), поэтому вы не можете изменить существующую строку. Лучшее, что можно сделать - создать новую строку, которая является вариантом оригинальной:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

Этот пример соединяет новую первую букву со срезом greeting, это не влияет на оригинальную строку.

12.6. Циклы и счет

Следующая программа подсчитывает количество вхождений буквы a в строку:

```
>>> word = 'banana'
>>> count = 0
>>> for letter in word:
if letter == 'a':
count = count + 1
>>> print count
3
```

Эта программа демонстрирует другой шаблон вычислений под названием - счетчик (`counter`). Переменной count присваивается нулевое начальное значение, затем это значение инкрементируется (увеличивается на 1) всякий раз, когда встречается символ a. Когда цикл завершается, в count содержится результат подсчета.

12.7. Оператор in

Слово `in` - это логический оператор, который принимает две строки и возвращает True, если первая строка является подстрой во второй строке:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

12.8. Сравнение строк

Оператор сравнения работает для строк. Рассмотрим вариант, если две строки одинаковые (`equal`):

```
>>> word = 'banana'
>>> if word == 'banana':
print 'All right, bananas.'
```

Другие операции сравнения полезны для ввода слова в алфавитном порядке:

```
word = 'banan'

if word < 'banana':
      print 'Your word, ' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word, ' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

Python не обрабатывает буквы верхнего и нижнего регистра так же как это делают люди. Все буквы верхнего регистра идут перед буквами нижнего регистра, так:

```
Your word, Pineapple, comes before banana.
```

Общий подход к решению этой проблемы - преобразовать строку в

стандартный формат, такой как нижний регистр, перед выполнением сравнения. Имейте это в виду, чтобы защитить себя от вооруженного Ананасом человека (Pineapple).

12.9. Строковые методы

Строка - пример объекта в Python. Объекты содержат данные (фактически саму строку) и методы, которые являются функциями, встроенными в объект и доступными для любого экземпляра (instance) объекта.

В Python есть функция dir, которая выводит список доступных методов для объекта. Функция type показывает тип объекта:

```
>>> stuff = 'Hello world'
>>> type(stuff)
< type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:
capitalize(...)
S.capitalize() -> string
Return a copy of the string S with only its first character
capitalized.
>>>
```

Вы можете воспользоваться функцией `help`, чтобы получить дополнительную информацию о методе. Лучший источник документации по строковым методам находится тут: ссылка: docs.python.org/library/string.html

Метод вызывается подобно функции - он принимает на вход аргументы и возвращает значение, но различается синтаксис. Мы вызываем метод путем добавления имени метода к имени переменной, используя точку в качестве разделителя.

Например, метод upper получает на вход строку и возвращает новую строку со всеми буквами в верхнем регистре:

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

Пустые круглые скобки означают, что метод не имеет аргументов. Вызов метода называется вызовом (invocation), в этом случае мы можем сказать, что вызвали метод upper объекта word.

Рассмотрим строковый метод find:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
>>>
```

В этом примере мы вызвали метод `find` объекта `word` и передали в качестве входного параметра букву, которую ищем. Метод `find` может искать подстроки, а не только отдельные символы:

```
>>> word.find('na')
2
```

В качестве второго аргумента метод `find` принимает индекс, с которого начинается поиск вхождения:

```
>>> word.find('na', 3)
4
```

Одной из распространенных задач является устранение пробелов (пробелов, табуляции или символов перевода строки) с самого начала и

до конца строки с помощью метода `strip`:

```
>>> line = ' Here we go '
>>> line.strip()
'Here we go'
```

Некоторые методы, такие как `startswith` возвращают логические значения:

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Предварительно применим метод `lower`, переводящий строку в нижний регистр:

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower()
'please have a nice day'
>>> line.lower().startswith('p')
True
```

В последнем примере мы вызвали в одном выражении подряд два метода.

12.10. Разбор (parsing) строк

Часто мы хотим заглянуть в строку и найти подстроку. Для примера, если мы имеем несколько строк, отформатированных следующим образом:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 200
```

И мы хотим вытащить только вторую часть адреса (например, uct.ac.za) из каждой строки. Это можно сделать с использованием метода `find` и

строкового среза.

Во-первых, мы находим положение at-символа (@) в строке. Затем находим позицию первого пробела после at-символа. После этого с использованием строкового среза извлекаем часть строки, которую ищем:

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5
09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ',atpos)
>>> print sppos
31
>>> host = data[atpos+1:sppos]
>>> print host
uct.ac.za
```

Мы используем версию метода find, которая позволяет нам задать позицию в строке, начиная с которой хотим выполнять поиск.

12.11. Оператор форматирования

Оператор форматирования % позволяет создавать строки, заменяя часть строки с данными, хранящимися в переменных. Когда он применяется к целым числам - это операция взятия по модулю.

Первым операндом является строка форматирования (format string), которая содержит одну или более последовательностей форматирования (format sequences), которые определяют форматирование второго оператора.

Результатом является строка.

Например, последовательность форматирования '%d' означает, что второй оператор должен быть отформатирован как целочисленное значение (`d` от `decimal`):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

Результатом является строка '42', которую не следует путать с целочисленным значением 42.

Последовательность форматирования может появиться в любой части строки, поэтому возможно встраивать последовательность в предложение:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

Если есть более чем одна форматируемая последовательность в строке, второй аргумент должен быть кортежем (tuple). Каждая форматируемая последовательность сочетается по порядку с элементом кортежа. В следующем примере используется '%d' форматирование для целочисленных значений, '%g' - для чисел с плавающей точкой (не спрашивайте, почему), и '%s' - для форматирования строки:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

Количество элементов в кортеже должно совпадать с количеством форматируемых последовательностей в строке. Кроме того, типы элементов должны соответствовать форматируемым последовательностям:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

Более подробно об операторе форматирования:

ссылка: docs.python.org/lib/typesseq-strings.html

12.12. Словарь

Счетчик (counter): A variable used to count something, usually initialized to zero and then incremented.

Пустая строка (empty string): A string with no characters and length 0, represented by two quotation marks.

Оператор форматирования (format operator): An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

Последовательность форматирования (format sequence): A sequence of characters in a format string, like %d, that specifies how a value should be formatted.

Строка форматирования (format string): A string, used with the format operator, that contains format sequences.

Флаг (fag): A boolean variable used to indicate whether a condition is true.

Вызов (invocation): A statement that calls a method.

Неизменяемый (immutable): The property of a sequence whose items cannot be assigned.

Индекс (index): An integer value used to select an item in a sequence, such as a character in a string.

Элемент (item): One of the values in a sequence.

method: A function that is associated with an object and called using dot notation.

Объект (object): Something a variable can refer to. For now, you can use "object" and "value" interchangeably.

Поиск (search): A pattern of traversal that stops when it finds what it is looking for.

Последовательность (sequence): An ordered set; that is, a set of values where each value is identified by an integer index.

Срез (slice): A part of a string specified by a range of indices.

Обход (traverse): To iterate through the items in a sequence, performing a similar operation on each.

Программа с вводом числа

Видео

Файлы

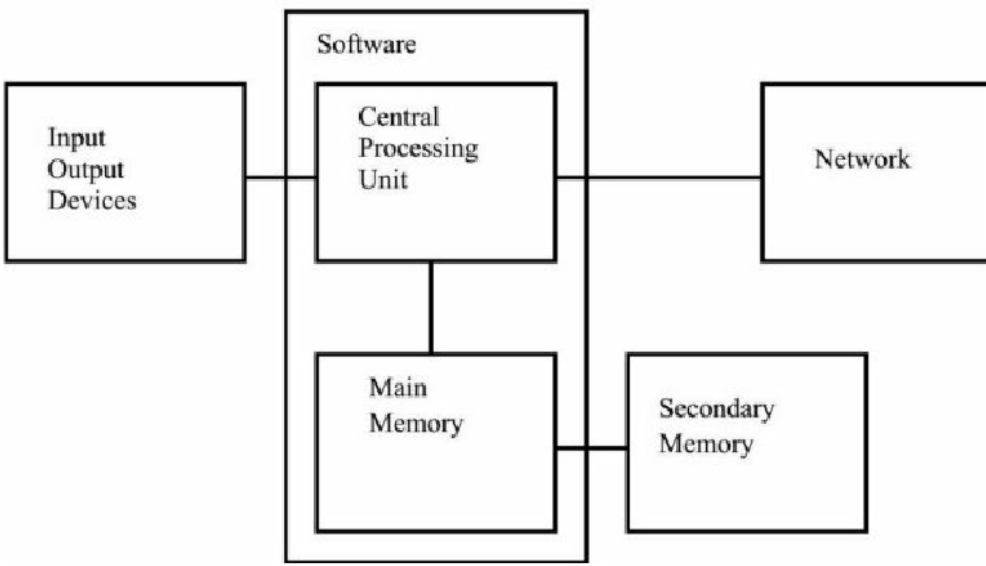
Видео

14.1. Введение

До сих пор мы учились тому, как писать программы и сообщать о наших намерениях центральному процессору (Central Processing Unit), используя условные инструкции, функции и итерации. Мы учились, как создавать и использовать структуры данных в оперативной памяти (Main Memory). В CPU и памяти работает и выполняется наше программное обеспечение (ПО).

Здесь происходит "мышление".

Но если вы помните из нашего обсуждения архитектуры оборудования, одно выключение питания и все хранящееся в CPU или оперативной памяти стирается. До этого момента наши программы имели временное развлекательное назначение для изучения Python.



В этой главе мы начинаем работать с вторичной памятью (Secondary Memory) или файлами. Вторичная память не очищается, когда выключается электричество. В случае с USB-флешками, данные можно

на них записать, после этого USB-флешки могут быть удалены из системы и перенесены в другую систему.

Мы сосредоточимся на чтении и записи текстовых файлов, таких, которые создаются в текстовом редакторе. Позже мы рассмотрим, как работать с файлами баз данных, такими, как бинарные файлы, специально разработанные для чтения и записи через ПО баз данных.

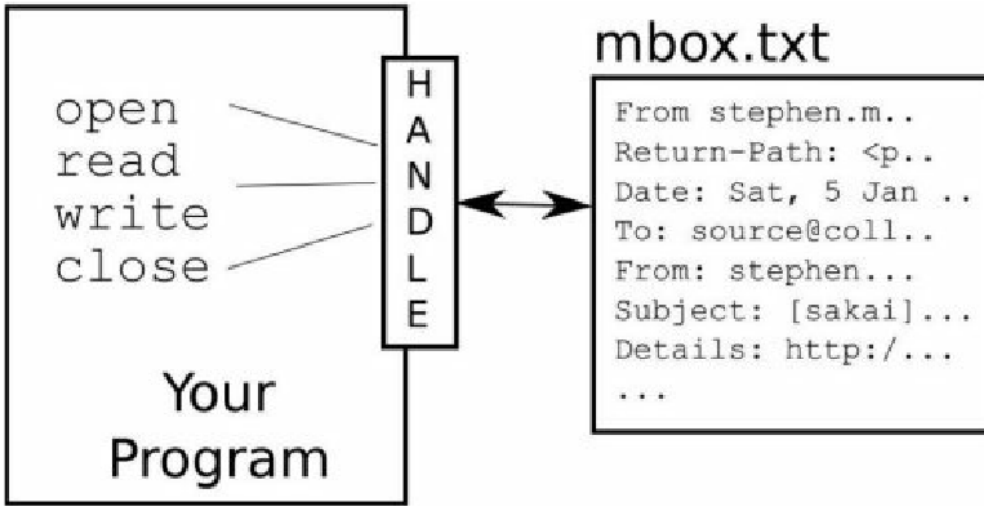
14.2. Открытие файлов

Когда мы хотим прочитать или записать файл (скажем, на жестком диске), во-первых мы должны открыть этот файл. Об открытии файла сообщается операционной системе (ОС), которая знает, где хранятся данные для каждого из файлов. Когда вы открываете файл, вы обращаетесь к ОС, чтобы найти файл по имени и удостовериться, что файл существует. В следующем примере мы открываем файл mbox.txt, который должен находиться в той же папке, откуда запускается Python. Скачать файл можно по ссылке:

ссылка: pycode.ru/files/python/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1251'>
```

Если функция open завершится успешно, то ОС вернет дескриптор файла (file handle). Дескриптор файла не является действующими данными, содержащимися в файле, это "обработчик", который мы можем использовать для чтения данных. У вас есть дескриптор, если запрашиваемый файл существует и имеет надлежащие права на чтение.



Если файл не существует, функция open выдаст ошибку с указанием причины и вы не получите дескриптор с доступом к содержимому файла:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
fhand = open('staff.txt')
IOError: [Errno 2] No such file or directory: 'staff.txt'
```

Позже мы воспользуемся инструкциями try и except, чтобы сделать более изящной обработку ситуации, когда мы открываем файл, который не существует.

14.3. Текстовый файл и строки

Текстовый файл можно представить как последовательность строк. В качестве примера рассмотрим текстовый файл, который содержит записи почтовой активности от различных лиц в команде разработчиков проекта с открытым исходным кодом:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
```

```
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

Полная версия файла доступна по ссылке:

ссылка: pycode.ru/files/python/mbox.txt и сокращенная версия

ссылка: pycode.ru/files/python/mbox-short.txt. Это файлы, которые содержат множество почтовых сообщений в стандартном формате. Строки, которые начинаются с "From " отделяют сообщение и строки, которые начинаются с "From:" являются частью сообщения. Более подробная информация по ссылке: ссылка: en.wikipedia.org/wiki/Mbox.

Чтобы разбить файл на строки, существует специальный символ, который представляет "конец строки" и называется символом новой строки.

В Python мы представляем символ новой строки, как строковую константу '/n'. Даже не смотря на то, что это выглядит как два символа - на самом деле один символ. Когда мы смотрим переменную, введя "stuff" в интерпретаторе, она отображается с \n в строке, но когда мы используем функцию print - видим, что строка разбилась на две по символу новой строки.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> len(stuff)
12
>>>
```

Также можно видеть, что длина строки 'Hello\nWorld!' составляет 12 символов, т.к. новый символ считается за один.

Поэтому, когда мы смотрим на строки в файле, необходимо представлять

(!), что есть специальные невидимые символы в конце каждой строки, которые обозначают конец строки.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008\n
Return-Path: <postmaster@collab.sakaiproject.org>\n
Date: Sat, 5 Jan 2008 09:12:18 -0500\n
To: source@collab.sakaiproject.org\n
From: stephen.marquard@uct.ac.za\n
Subject: [sakai] svn commit: r39772 - content/branches/\n
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772\n
...
```

Таким образом, символ новой строки отделяет символы в файле внутри строк.

14.4. Чтение файлов

До тех пор, пока дескриптор файла не содержит данные, создадим конструкцию с оператором for, читая в цикле и увеличивая счетчик для каждой строки в файле.

```
>>> fhand = open('mbox.txt')
>>> count = 0
>>> for line in fhand:
count = count + 1
>>> print('Line Count:', count)
Line Count: 132045
```

Мы можем использовать дескриптор файла как последовательность в цикле for. Наш цикл считает количество строк в файле и выводит окончательное значение на экран. Грубый перевод цикла for на человеческий язык: "для каждой строки в файле, представленной файловым дескриптором, увеличить переменную count на единицу".

Причина, по которой функция open не читает весь файл, заключается в том, что файл может иметь гигабайты данных. Функция open занимает одинаковое количество времени, независимо от размера файла.

В случае, когда файл читается с использованием цикла for, Python

заботится о разбиении данных в файле на отдельные строки, используя символ новой строки. Python читает каждую строку, используя новую строку, и включает новую строку, как последний символ в переменную line для каждой итерации цикла for.

Поэтому цикл for читает данные по одной строке за раз, это эффективное чтение и подсчет строк в большом файле без загрузки всей памяти для хранения данных. Программа, написанная выше, может считать строки в файле любого размера, используя немного памяти для чтения каждой строки, подсчета и их сброса.

Если вы знаете что, файл является относительно небольшим по сравнению с размером вашей оперативной памяти, вы можете прочитать весь файл в одну строку, используя метод read для дескриптора файла.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

В этом примере все содержимое (94626 символов) файла mbox-short.txt прочитано непосредственно в переменную inp. Мы воспользовались строковым срезом для печати 20 символов строки данных, хранящейся в inp.

Помните, что такой способ использования функции open возможен только, если файл данных будет удобно помещаться в оперативную память вашего компьютера, иначе используйте циклы for или while.

14.5. Поиск через файл

Когда вы ищете данные через файл, это очень общий шаблон чтения через файл, игнорируя большинство строк и обрабатывая строки, которые соответствуют определенному критерию. Мы можем объединить шаблон для чтения файла со строковыми методами, построив простой механизм поиска.

К примеру, если мы хотим прочитать файл и вывести на экран строки, которые начинаются с префикса "From:", мы можем воспользоваться строковым методом startswith, выбрав строки с желанным префиксом.

```
>>> fhand = open('mbox-short.txt')
>>> for line in fhand:
if line.startswith('From:') :
print(line)
```

Программа выполняется, получаем следующий результат:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
...
```

Мы получили желаемый результат. Но откуда пустые строки? Это результат невидимости символа новой строки. Каждая строка оканчивается новой строкой, т.к. функция print печатает строку из переменной line, которая включает новую строку и затем функция print добавляет другую новую строку, в результате мы наблюдаем эффект двойного пробела.

Мы могли бы использовать строковый срез, чтобы напечатать все, кроме последнего символа, но простой подход заключается в использовании метода rstrip, который удаляет пробелы в правой части строки следующим образом:

```
>>> for line in fhand:
        line = line.rstrip()
        if line.startswith('From:'):
            print(line)
```

Получим следующий результат:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
```

```
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
...
```

Вы можете усложнить структуру поиска, воспользовавшись инструкцией continue. Основная идея поиска в цикле - вы ищете "интересные" строки и пропускаете "неинтересные". И когда находите интересную строку - выполняете с ней какие-то действия.

Мы можем построить цикл по следующему шаблону, пропуская неинтересные строки:

```
>>> for line in fhand:
        line = line.rstrip()
        if not line.startswith('From:'):
            continue
        print(line)
```

Вывод программы совпадает с предыдущим.

Мы можем использовать строковый метод find для имитации поиска в текстовом редакторе. Так как find ищет вхождения строки в другой строке и либо возвращает позицию строки или -1, если строка не найдена, мы можем написать следующий цикл, чтобы показать строки, которые содержат "@uct.ac.za" (то есть письма приходят из университета Кейптауна в Южной Африке):

```
>>> fhand = open('mbox-short.txt')
>>> for line in fhand:
        line = line.rstrip()
        if line.find('@uct.ac.za') == -1 :
            continue
        print(line)
```

Получили следующий результат:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to
stephen.marquard@uct.ac.za using -f
```

From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan 4 07:02:32 2008
X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
r39753 | david.horwitz@uct.ac.za | 2008-01-04 13:05:51 +0200 (Fri, 04 Jan 2008) | 1 line
From david.horwitz@uct.ac.za Fri Jan 4 06:08:27 2008
X-Authentication-Warning: nakamura.uits.iupui.edu: apache set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
...

Печать файла

Видео

Списки

Видео

16.1. Список является последовательностью

Подобно строке, список - последовательность значений. В строке значениями являются символы, в списке - они могут быть любого типа. Значения в списке называются элементами (elements) или иногда записями (items).

Есть несколько путей создания нового списка, самый простой - заключить элементы в квадратные скобки [ ]:

```
>>> [10, 20, 30, 40]
[10, 20, 30, 40]
>>> ['crunchy frog', 'ram bladder', 'lark vomit']
['crunchy frog', 'ram bladder', 'lark vomit']
```

В первом примере - список из четырех целых чисел, во втором - список из трех строк.

Элементы списка могут быть разного типа. Следующий список содержит строку, число с плавающей точкой, целое число и (внимание!) другой список:

```
>>> ['spam', 2.0, 5, [10, 20]]
['spam', 2.0, 5, [10, 20]]
```

Список внутри другого списка является вложенным (nested).

Список, который не содержит элементом, называется пустым (empty list), его можно создать с помощью пустых квадратных скобок [ ].

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

16.2. Списки изменчивы

Синтаксис для доступа к элементам списка похож на доступ к символам строки - оператор скобки. Выражение внутри скобок определяют индекс. Запомните, что индекс начинается с нуля:

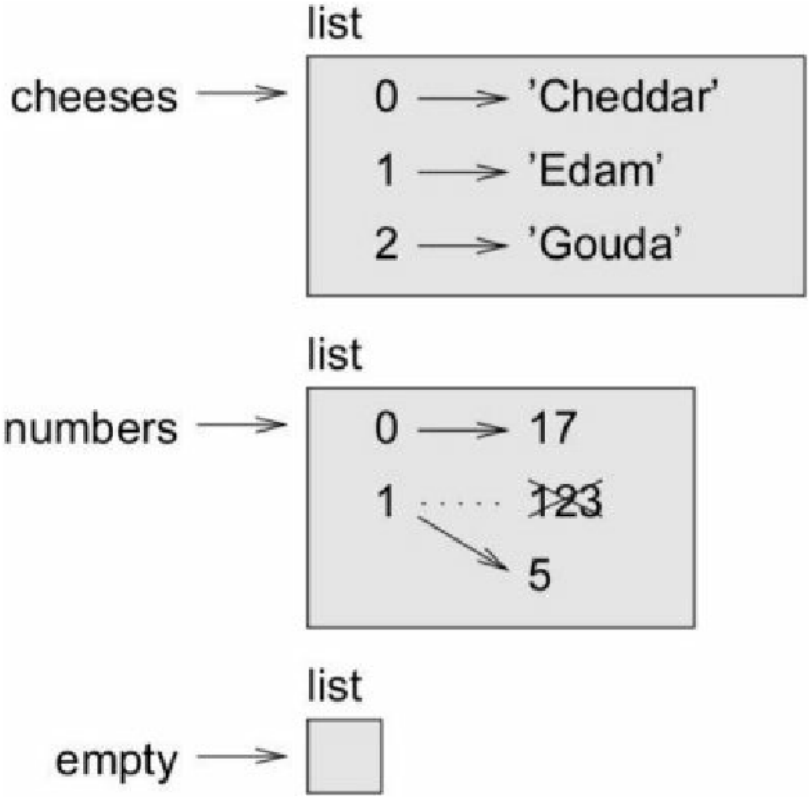
```
>>> print cheeses[0]
Cheddar
```

В отличие от строк, списки можно изменять, поэтому вы можете изменить порядок элементов в списке или переприсвоить элементы в списке. Когда оператор скобок появляется в левой части выражения, он определяет элемент списка, который будет присвоен.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

Вы можете представить список как связь между индексами и элементами. Эта связь называется отображением (mapping), каждый индекс отображается на один из элементов.

Следующая диаграмма состояний показывает списки cheeses, numbers и empty:



Списки на рисунке представлены прямоугольниками со словом список (list) снаружи и элементами списка внутри.

Список индексов работает так же, как и строковые индексы:

- Любое целочисленное выражение можно использовать в качестве индекса.
- Если вы попытаетесь прочитать или записать элемент, которого не существует, вы получите IndexError.
- Если индекс имеет отрицательное значение, он ведет счет в обратном направлении от конца списка.

Оператор in работает аналогично для списков.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
```

```
True
>>> 'Brie' in cheeses
False
```

16.3. Обход списка

Наиболее общий путь обхода элементов списка - использование цикла for.

Похожий синтаксис используется для обхода строк:

```
>>> for cheese in cheeses:
        print cheese


Cheddar
Edam
Gouda
```

Это замечательно работает, если вам необходимо прочитать элементы списка. Но если вы хотите записать или обновить элементы, понадобятся индексы. Общий подход, чтобы это сделать - объединить функции range и len:

```
>>> for i in range(len(numbers)):
          numbers[i] = numbers[i] * 2

>>> print numbers
[34, 10]
```

Этот цикл обходит список и обновляет каждый элемент. len возвращает число элементов в списке. range возвращает список индексов от 0 до n-1, где n - длина списка. За каждый шаг в цикле переменной i присваивается индекс следующего элемента. Выражение присваивания в теле цикла использует i для чтения старого значения элемента и записи нового значения.

При обходе пустого списка в цикле for никогда не выполнится тело цикла:

```
>>> for x in empty:
        print 'This never happens.'
```

Хотя список может содержать другой список, вложенный список считается одним элементом. Длина списка будет равна четырем:

```
>>> len(['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]])
4
```

16.4. Операторы списка

Оператор + объединяет списки:

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Аналогичным образом, оператор * повторяет список заданное число раз:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1,2,3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

16.5. Срез списка

Оператор среза также работает для списков:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Если опустить первый индекс, то срез начнется с начала. Если опустить второй - срез закончится в конце. Если опустить оба значения, то срез будет являться копией всего списка.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Т.к. списки можно изменять то, часто создается копия перед выполнением операций, которые складывают, удлиняют или искажают списки.

Оператор среза в левой части выражения может обновить несколько элементов:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x','y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

16.6. Методы списков

Python предоставляет методы, которые оперируют со списками. Например,

append добавляет новый элемент в конец списка:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
>>>
```

extend принимает в качестве аргумента список и добавляет все элементы:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

sort организует элементы списка от низших к высшим:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

Большинство методов списка не имеют типа (void), они изменяют список и возвращают None. Если вы случайно напишете t = t.sort(), то результат вас разочарует.

16.7. Удаление элементов

Существует несколько подходов для удаления элементов из списка. Если известен индекс элемента, можно воспользоваться pop:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
>>>
```

pop изменяет список и возвращает элемент, который был удален. Если не указан индекс, то удаляется и возвращается последний элемент списка.

Если вам не нужно удалять значение, можно воспользоваться оператором del:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

Если вы знаете элемент, который хотите удалить (но не его индекс), можно воспользоваться remove:

```
>>> t = ['a', 'b', 'c']
```

```
>>> t.remove('b')
>>> print t
['a', 'c']
```

remove возвращает значение None.

Для удаления больше, чем одного элемента, можно использовать del со срезом индекса:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

16.8. Списки и функции

Существует несколько встроенных функций, которые могут быть использованы для списков, они позволяют быстро просмотреть список без написания собственного цикла:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25.666666666666668
```

Функция sum() работает только, когда все элементы списка числовые. Другие функции (max(), len() и т.д.) работают со списками строк и другими типами, которые могут быть сопоставимы.

Мы можем переписать программу, которая рассчитывала среднее значение чисел в списке с использованием списка.

Сначала программа, подсчитывающая среднее значение, без списка:

```
total = 0
count = 0
while ( True ) :
      inp = raw_input('Enter a number: ')
      if inp == 'done' : break
      value = float(inp)
      total = total + value
      count = count + 1
      average = total / count
print 'Average:', average
```

Можем просто запомнить каждое число, которое вводит пользователь и с использованием встроенной функции подсчитать sum и count в конце.

```
numlist = list()
while ( True ) :
      inp = raw_input('Enter a number: ')
      if inp == 'done' : break
      value = float(inp)
      numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

Мы создали пустой список перед началом цикла и затем каждый раз добавляли число в этот список.

16.9. Списки и строки

Строка - последовательность символов, список - последовательность значений, но список символов - это не тоже самое, что строка. Преобразовать строку в список символов можно с помощью list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

Т.к. list - это имя встроенной функции, вы должны избегать его использование в качестве имени переменной. Я избегаю использовать l, т.к. это похоже на 1, поэтому я использую t.

Функция list разбивает строку на отдельные буквы. Если вы хотите разбить строку на слова, вы можете использовать метод split:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

При вызове split можно передать аргумент, который задает разделитель (delimiter), вместо пробела по умолчанию:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join - противоположность split, он принимает список строк и объединяет элементы.

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

В этом случае разделителем является пустая строка, поэтому join помещает пробелы между словами. Чтобы соединить строки без пробелов, в качестве разделителя необходимо использовать пустую строку.

16.10. Разбор списков

Обычно, когда мы читаем файл, мы хотим что-то сделать, а не просто вывести на экран всю строку. Часто мы хотим найти "интересные

строки", а затем разобрать строки, чтобы найти некоторые интересные части строки. Как быть, если мы хотим распечатать день недели у тех строк, которые начинаются с "From ".

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Метод split очень эффективен, когда сталкивается с подобной проблемой. Мы можем написать небольшую программу, которая ищет строки, начинающиеся с "From ", а затем разделить (split) эти строки и распечатать третье слово в строке:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if not line.startswith('From ') : continue
        words = line.split()
        print words[2]
```

Мы используем условие if, которое пропускает все строки, начинающиеся не с 'From '.

Результат работы программы будет следующим:

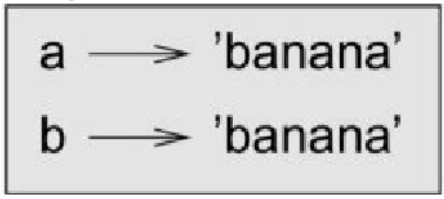
```
Sat
Fri
Fri
Fri
Fri
Fri
...
```

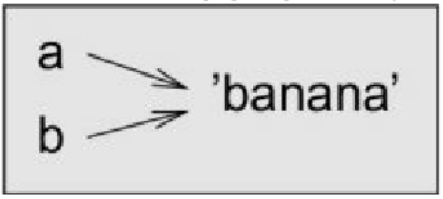
16.11. Объекты и значения

Если мы выполним эти операторы присваивания:

```
a = 'banana'
b = 'banana'
```

Мы знаем, что a и b ссылаются на строку, но мы не знаем, ссылаются ли они на одну и ту же строку. Возможны два варианта:





В первом случае, a и b ссылаются на два различных объекта, которые имеют некоторое значение. Во втором случае, они ссылаются на один и тот же объект.

Чтобы проверить, ссылаются ли две переменные на один и тот же объект, вы можете использовать оператор is.

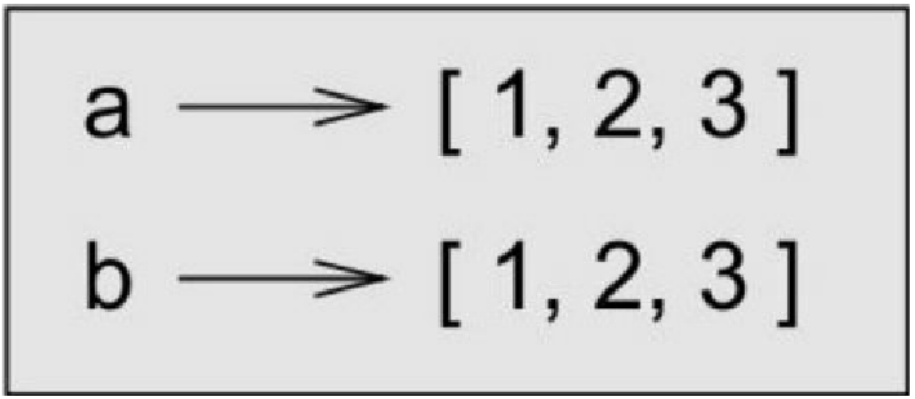
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

В этом примере Python создает только один строковый объект, a и b ссылаются на него.

Но когда вы создаете два списка, вы получите два объекта:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

На диаграмме это выглядит следующим образом:



В этом случае мы можем сказать, что два списка эквивалентны (equivalent), т.к. имеют одинаковые элементы, но не идентичны (identical), т.к. это не один и тот же объект. Если два объекта идентичны, они также эквивалентны, но если они эквивалентны не обязательно, что они идентичны.

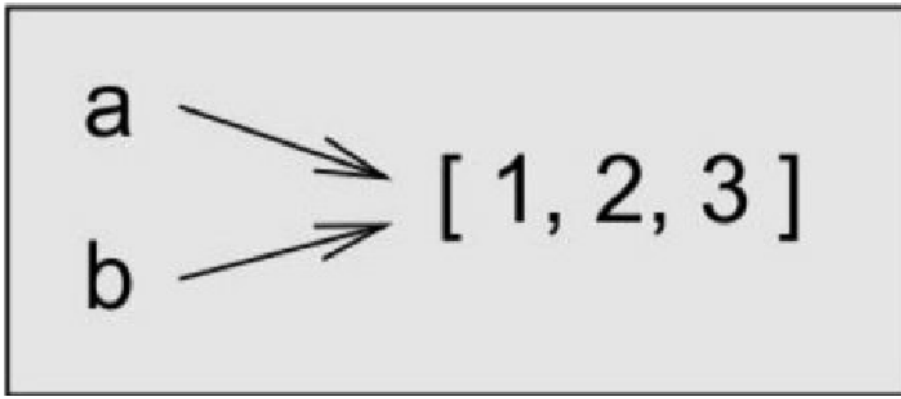
До этого момента, мы использовали понятия 'объект' (object) и 'значение' (value), как синонимы, но более точно говорить, что объект имеет значение. Если вы выполните a = [1,2,3], то переменная a ссылается на объект-список, значением которого является определенная последовательность элементов.

16.12. Псевдонимы (Aliasing)

Если a ссылается на объект, и вы присваиваете b = a то, затем обе переменные ссылаются на один и тот же объект:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Схема выглядит следующим образом:



Ассоциация переменной с объектом называется ссылкой (reference). В этом примере две ссылки на один объект. Объект с более чем одной ссылкой имеет больше одного имени, поэтому мы говорим, что объект имеет псевдонимы (aliased).

Если псевдоним объекта изменяется, то эти изменения касаются других псевдонимов:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Хотя такое поведение может быть полезным, существует вероятность ошибиться. В общем, более безопасно избегать псевдонимов при работе с изменяемыми объектами.

Для неизменяемых объектов, таких как строки, псевдонимы не являются большой проблемой.

16.13. Список аргументов

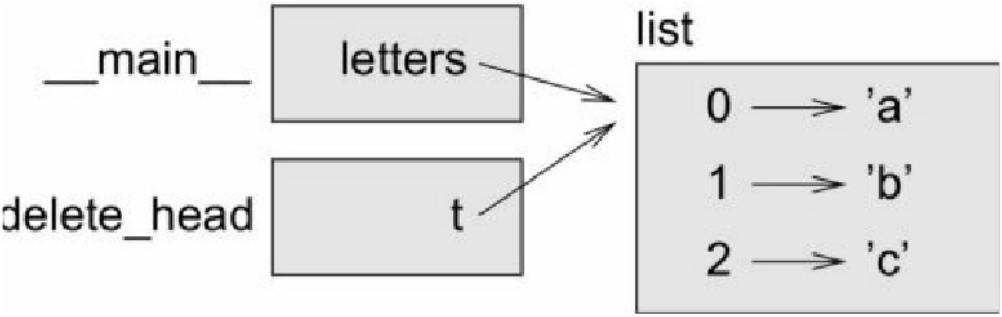
Когда вы передаете список в функцию, функция получает ссылку на список.

Если функция изменяет параметр-список, вызывающий видит изменения.

Например, delete_head удаляет первый элемент из списка:

```
>>> def delete_head(t):
del t[0]
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

Параметр t и переменная letters - псевдонимы для одного и того же объекта. Схема стека выглядит следующим образом:



Поскольку список является общим для двух фреймов, я изобразил его между ними.

Важно различать операции, изменяющие списки и операции, которые создают новые списки. Например, метод append изменяет список, оператор + создает новый список:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None
>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3, 3]
>>> t2 is t3
False
```

Это различие очень важно, когда вы пишете функции, которые должны изменять списки. Например, эта функция не удаляет первый элемент списка:

```
def bad_delete_head(t):
    t = t[1:] # WRONG!
```

Оператор среза создает новый список и присваивает t указатель на него, но это никак не отражается на списке, который был передан в качестве аргумента.

В качестве альтернативы можно написать функцию, которая создает и

возвращает новый список. Например, tail возвращается все, кроме первого элемента списка:

```
def tail(t):
    return t[1:]
```

Эта функция оставляет первоначальный список без изменений. Вот как она используется:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

16.15. Словарь

aliasing: A circumstance where two or more variables refer to the same object.

delimiter: A character or string used to indicate where a string should be split.

element: One of the values in a list (or other sequence), also called items.

equivalent: Having the same value.

index: An integer value that indicates an element in a list.

identical: Being the same object (which implies equivalence).

list: A sequence of values.

list traversal: The sequential accessing of each element in a list.

nested list: A list that is an element of another list.

object: Something a variable can refer to. An object has a type and a value.

reference: The association between a variable and its value.

Поиск строки

Видео

Словари

Видео

Словарь (dictionary) похож на список, но имеет более широкие возможности. В списке позиция (или индекс) имеет целочисленное значение, в словаре индекс может быть (почти) любого типа.

Вы можете представить словарь как отображение между множеством индексов (тут они называются ключами) и множеством значений. Каждый ключ (key) отображается на значение. Связь между ключом и значением называется парой ключ-значение (key-value pair) или иногда записью (item).

В качестве примера, мы создадим словарь, который отображает английские и испанские слова, таким образом ключи и значения являются строками.

Функция dict создает новый словарь без записей. Т.к. dict - имя встроенной функции, вы должны исключить его из имен переменных.

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

Фигурные кавычки означают пустой словарь. Для добавления записей в словарь, можно использовать квадратные скобки:

```
>>> eng2sp['one'] = 'uno'
```

Эта строка создает запись, которая отображает ключ 'one' на значение 'uno'. Если мы распечатаем словарь снова, то увидим пару ключ-значение, разделенную двоеточием:

```
>>> print eng2sp
{'one': 'uno'}
```

Этот выходной формат совпадает с входным форматом. Например, можно создать новый словарь с тремя записями:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Если вы выведете на экран содержимое переменной eng2sp, то удивитесь:

```
>>> print eng2sp
{'three': 'tres', 'two': 'dos', 'one': 'uno'}
```

Порядок пар ключ-значение не совпадает. Фактически, если вы наберете этот же пример на своем компьютере, то можете получить совершенно другой результат. Порядок записей в словаре непредсказуем.

Но это не проблема, т.к. элементы словаря никогда не индексируются числовыми индексами. Вместо этого вы используете ключи для поиска соответствующих значений:

```
>>> print eng2sp['two']
dos
```

Ключ 'two' всегда отображается на значение 'dos', поэтому порядок записей не имеет значения.

Если ключ отсутствует в словаре, вы получите исключение:

```
>>> print eng2sp['four']
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
print eng2sp['four']
KeyError: 'four'
```

Функция len работает для словарей, она возвращает число пар ключ-значение:

```
>>> len(eng2sp)
3
```

Оператор in работает для словарей, он сообщает о похожих ключах в словаре.

```
>>> 'one' in eng2sp
```

```
True
>>> 'uno' in eng2sp
False
```

Чтобы убедиться, что какое-то значение встречается в словаре, вы можете использовать метод values, который возвращает значения в виде списка, а затем использовать оператор in:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

Оператор in использует различные алгоритмы для списков и словарей. Для списков он использует алгоритм линейного поиска. Время поиска пропорционально длине списка.

Для словарей Python использует алгоритм хеш-таблиц (hash table), который имеет замечательные свойства; in оператор занимает примерно столько же времени, сколько записей есть в словаре. Я не буду объяснять, почему хэш- функции, такие хорошие, но вы можете узнать об этом подробнее: ссылка: wikipedia.org/wiki/Hash_table

19.1. Словарь как набор счетчиков

Предположим, вы хотите подсчитать, сколько раз каждая буква встречается в строке.

Есть несколько способов, чтобы это сделать:

1. Вы можете создать 26 переменных, по одной на каждую букву алфавита. Затем можно обойти строку и для каждого символа увеличивать соответствующий счетчик на единицу, возможно с использованием условных выражений (chained conditional).

2. Вы можете создать список из 26 элементов. Затем можно перевести каждый символ в число (с помощью встроенной функцию ord), используя число как индекс в списке и инкрементируя соответствующий счетчик.

3. Вы можете создать словарь с символами в виде ключей и счетчиками в виде значений. При первой встрече символа, вы должны добавить запись в словарь. После этого можно инкрементировать значение существующей записи.

Каждый из этих вариантов выполняет одинаковые вычисления, но каждый из них реализует разные способы вычислений.

Реализация (implementation) - это способ выполнения вычислений, некоторые реализации лучше остальных. Например, преимущество реализации через словарь в том, что мы не должны знать заранее, какие буквы появятся в строке, у нас есть только свободное место для букв, которые появятся.

Код может выглядеть следующим образом:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c] + 1
    return d
```

Имя функции - гистограмма, т.к. это статистический термин для множества счетчиков (или частот).

Первая строка функции создает пустой словарь. Цикл for обходит строку. Всякий раз в цикле, если символ c не встречается в словаре, мы создаем новую запись с ключом c и присваиваем начальное значение 1 (т.к. мы встретили эту букву один раз). Если c уже есть в словаре, то мы инкрементируем d[c].

Так это работает:

```
>>> h = histogram('brontosaurus')
>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

19.2. Словари и файлы

Одно из распространенных применений словаря заключается в подсчете вхождений слов в текстовом файле. Давайте начнем с очень простого файла слов, взятого из текста Ромео и Джульетты:

ссылка: http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html

В первом множестве примеров мы будем использовать сокращенную и упрощенную версию текста без знаков препинания. Позже мы будем работать с текстом сцены, включающим знаки препинания.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Мы напишем программу на Python, которая читает строки файла, разбивает каждую строку на список слов, затем проходит в цикле через каждое слово в строке и с использованием словаря подсчитывает слова.

Вы увидите, что получится два цикла for. Внешний цикл для строк файла и внутренний цикл для каждого слова в отдельной строке. Это пример шаблона, который называется вложенные циклы (nested loops), т.к. один из циклов внешний (outer), а второй - внутренний (inner).

Комбинация из двух вложенных циклов гарантирует, что мы будем считать каждое слово в каждой строке входного файла.

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

counts = dict()
for line in fhand:
    words = line.split()
```

```
        for word in words:
            if word not in counts:
                counts[word] = 1
            else:
                counts[word] += 1
    print counts
```

Когда мы запускаем программу, мы видим сырой дамп (raw dump) всех счетчиков в неотсортированном хэш порядке (файл romeo.txt file доступен по адресу: ссылка: pycode.ru/files/python/romeo.txt))

```
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1, 'is': 3,
'through': 1, 'pale': 1,

'yonder': 1, 'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1,

'sick': 1, 'east': 1, 'breaks': 1, 'grief': 1, 'with': 1,
'light': 1, 'It': 1,

'Arise': 1, 'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

Немного неудобно просматривать словарь в поисках наиболее употребительных слов и их количества, поэтому нам нужно добавить еще несколько строк кода, чтобы получить более полезный результат на выходе.

19.3. Циклы и словари

Если вы используете словарь как последовательность (sequence) в операторе for, происходит обход ключей словаря. Например, print_hist выводит на экран каждый ключ и соответствующее ему значение:

```
def print_hist(h):
    for c in h:
        print c, h[c]
```

Результат имеет следующий вид:

```
>>> h = histogram('parrot')
>>> print_hist h
a 1
p 1
r 2
t 1
o 1
```

Снова ключи не имеют порядка.

Если вы хотите вывести на экран ключи в алфавитном порядке, сначала составьте список ключей в словаре с помощью метода словаря keys, затем отсортируйте этот список и в цикле, просматривая каждый ключ, выводите на экран пару ключ/значение, как показано далее:

```
def print_sorted_hist(h):
    lst = h.keys()
    lst.sort()
    for c in lst:
        print c, h[c]
```

Результат будет следующий:

```
>>> h = histogram('parrot')
>>> print_sorted_hist(h)
a 1
o 1
p 1
r 2
t 1
```

Теперь ключи отсортированы в алфавитном порядке.

19.4. Расширенный текстовый поиск

9.6. Словарь

dictionary: A mapping from a set of keys to their corresponding values.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

histogram: A set of counters.

implementation: A way of performing a computation.

item: Another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

key-value pair: The representation of the mapping from a key to a value.

lookup: A dictionary operation that takes a key and finds the corresponding value.

nested loops: When there is one or more loops "inside" of another loop. The inner loop runs to completion each time the outer loop runs once.

value: An object that appears in a dictionary as the second part of a key-value pair.

This is more specific than our previous use of the word "value."

Поиск популярных слов

Видео

Кортежи (tuples)

Видео

20.1. Кортежи неизменяемы

Кортеж [1] – это последовательность значений, аналогичная списку. Значения, хранимые в кортеже, могут быть различных типов; они индексируются целыми числами. Важное отличие от списков – кортежи неизменяемы.

Также кортежи сравниваются и хешируются, что позволяет сортировать их списки и использовать кортежи в качестве ключевых значений в словарях Питона. Синтаксически, кортеж – это список значений, разделенных запятыми:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Кортежи обычно заключают в круглые скобки, хотя это и не обязательно. Скобки помогают опознать кортеж в исходном коде Питона:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

При создании кортежа с одним элементом необходимо в конце поставить запятую:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Без запятой в конце Питон распознает ('a') как выражение в скобках, которое является строкой:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Другой способ образования кортежа – с помощью встроенной функции tuple. При вызове без аргумента она создает пустой кортеж:

```
>>> t = tuple()
>>> print t
()
```

Если аргумент является последовательностью (строкой, списком или кортежем), то результатом вызова функции tuple станет кортеж с последовательностью элементов:

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Так как tuple является именем конструктора, следует избегать использования этого слова в качестве переменной.

Большинство операторов списка также работают и с кортежами.

Оператор квадратные скобки индексирует элемент:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

Оператор двоеточие задает диапазон элементов:

```
>>> print t[1:3]
('b', 'c')
```

При попытке изменить один из элементов кортежа выдается сообщение об ошибке:

```
>>> t[0] = 'A'

TypeError: object doesn't support item assignment
```

(Ошибка Типа: объект не допускает изменения его элементов)

Нельзя изменять элементы кортежа, можно лишь заменить один кортеж на другой:

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

20.2. Сравнение кортежей

Операторы сравнения работают как с кортежами, так и с другими последовательностями: Питон сначала сравнивает первые элементы последовательностей; если они равны, сравниваются следующие элементы и так до тех пор, пока не найдутся неравные элементы. Последующие элементы не рассматриваются (даже если они очень большие).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Функция сортировки работает аналогичным образом. Сначала она сортирует по первым элементам, в случае равенства первых элементов — по вторым и т.д.

Эта характерная особенность сортировки легла в основу метода под названием DSU (Decorate-Sort-Undecorate): разметка (Decorate) последовательности с помощью создания списка кортежей, каждый из которых включает элемент исходной последовательности плюс один или несколько вспомогательных ключей сортировки, предшествующих этому элементу; затем сортировка (Sort) списка кортежей с помощью встроенной функции Питона и далее удаление разметки (Undecorate) путем извлечения отсортированных элементов исходной последовательности.

Например, предположим, что есть список слов, которые необходимо отсортировать по их длине – от самого длинного к самому короткому:

```
def sort_by_length(words):
t = list()
for word in words:
t.append((len(word), word))
t.sort(reverse=True)
res = list()
for length, word in t:
res.append(word)
return res
```

Первый цикл создаст список кортежей, где каждый кортеж – это слово с определенной длиной.

При сортировке сравниваются первые элементы — длины слов, вторые элементы рассматриваются лишь при равенстве первых. Ключевой аргумент reverse=True предписывает выполнять сортировку в порядке убывания.

Второй цикл проходит список кортежей и формирует список слов в порядке убывания по длине.

20.3. Присваивание кортежей

Одной из особенностей синтаксиса языка Питон является возможность размещать кортеж слева от оператора присваивания. Это позволяет за один раз присваивать значение более чем одной переменной.

В примере мы используем двухэлементный список (который представляет собой последовательность) и присваиваем первый и второй его элементы переменным x и y в одном операторе присваивания.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
```

```
'have'
>>> y
'fun'
>>>
```

Здесь нет ничего загадочного, Питон просто преобразует синтаксис присвоения кортежа в следующий код[2]:

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

Стилистически, когда кортеж ставится с левой стороны оператора присваивания, скобки обычно опускаются, но следующий код в той же степени допустим:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

Хитроумное использование присваивания кортежей позволяет нам поменять местами значения двух переменных в одном операторе:

```
>>> a, b = b, a
```

Обе части этого оператора – кортежи, но слева – кортеж переменных, а справа – кортеж выражений. Каждое значение справа присваивается

соответствующей переменной слева. Все выражения справа вычисляются до любых присваиваний.

Количество переменных слева должно быть равно количеству значений справа:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
(Ошибка Значений: слишком много значений для распаковки)
```

В общем случае справа может быть любой тип последовательности (строка, список или кортеж). Например, чтобы разделить электронный адрес на имя пользователя и название домена, можно использовать фрагмент кода:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Возвращаемое значение метода split является списком из двух элементов; первый элемент присваивается переменной uname, второй — переменной domain.

```
>>> print uname
monty
>>> print domain
python.org
```

20.4. Словари и кортежи

У словарей есть метод с названием "items", который возвращает список кортежей, где каждый кортеж – это пара ключ-значение[3]:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> print t
[('a', 10), ('c', 22), ('b', 1)]
```

Как и следовало ожидать, в словаре элементы располагаются в произвольном порядке. Но, поскольку список кортежей является списком и кортежи можно сравнивать — можно сортировать список кортежей. Преобразование словаря в список кортежей – это способ вывода содержимого словаря, отсортированного по ключу:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> t
[('a', 10), ('c', 22), ('b', 1)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

Новый список сортируется в алфавитном порядке по значению ключа.

20.5. Множественное присваивание при работе со словарями

Сочетая использование функции items, присваивание кортежей и цикл for, можно написать изящный фрагмент кода для перечисления ключей и их значений в словаре в одном цикле:

```
for key, val in d.items():
print val, key
```

В этом цикле используются две итерационные переменные, потому что функция items возвращает список кортежей; пара переменных key, val также образует кортеж, который пробегает все пары ключ/значение, содержащиеся в словаре.

После каждой итерации переменные key и val переходят к следующей паре ключ/значение, содержащейся в словаре (в порядке, задаваемой хеш-функцией).

На выходе цикла получим:

```
0 a
2 c
1 b
```

Значения напечатаны в порядке, задаваемом хеш-функцией (т.е. без определенной упорядоченности).

Сочетая два указанных метода, мы можем вывести содержание словаря, которое отсортировано по значениям, содержащимся в парах ключ/значение.

Для этого сначала создается список кортежей, где каждый кортеж представляет собой пару (значение, ключ).Метод items даст нам список кортежей вида (ключ, значение), однако на этот раз мы хотим выполнить сортировку по значениям, а не по ключам.

Когда список кортежей (значение, ключ) создан, несложно отсортировать его в обратном порядке и напечатать новый отсортированный список.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
... l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

Создав вручную список кортежей, в которых значение ключа является первым элементом, мы можем отсортировать список и получить содержимое словаря, отсортированное по значениям ключей.

20.6. Наиболее часто встречающиеся слова

Вернемся назад к нашему примеру – отрывку текста из произведения "Ромео и Джульетта", действие 2, сцена 2. Мы можем дополнить нашу программу, если применим технику вывода десяти наиболее распространенных в тексте слов, используя следующий код:

```
import string
fhand = open('romeo-full.txt')
counts = dict()

for line in fhand:

line = line.translate(None, string.punctuation)
line = line.lower()
words = line.split()

for word in words:
if word not in counts:

counts[word] = 1
else:
counts[word] += 1
# Sort the dictionary by value
lst = list()
for key, val in counts.items():
lst.append( (val, key) )
lst.sort(reverse=True)
for key, val in lst[:10] :
print key, val
```

Остается неизменной первая часть программы, которая читает файл и создает словарь, сопоставляющий каждому слову в документе количество его вхождений в текст. Но вместо обычного вывода количества вхождений и завершения программы мы создаем список кортежей (значение, ключ) и затем сортируем его в порядке убывания.

Поскольку значение ключа является первым элементом кортежа, оно будет использоваться первым для сравнения; если кортежей с подобным значением несколько, то будут сравниваться вторые элементы кортежей,

т.е. ключи. Таким образом, кортежи с одинаковыми значениями будут сортироваться по ключам в алфавитном порядке.

В конце мы запишем изящный цикл for, который выполняет множественное присваивание в каждой итерации и выводит десять наиболее распространенных слов с помощью перебора части списка (lst[:10]).

Вывод наконец-то выглядит так, как это требуется для анализа частоты слов.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

То, что эту непростую задачу разбора и анализа данных можно решить с помощью небольшой и понятной программы на Питоне из 19 строк, подтверждает, что Питон является очень удобным языком программирования в применении к задачам исследования информации.

20.7. Использование кортежей в качестве ключей в словарях

Кортежи, в отличие от списков, хешируемы, поэтому, если мы хотим создать составной ключ для использования в словаре, нужно использовать кортеж в качестве ключа.

Например, нам понадобится составной ключ, если мы захотим создать телефонный справочник, который отображает пары (фамилия, имя) на телефонные номера. Предположим, что переменные last (фамилия – last name), first (имя – first name) и number (номер) определены, тогда мы

можем внести запись в справочник следующим образом:

```
directory[last,first] = number
```

Выражение в скобках является кортежем. Мы можем использовать присваивание кортежей в цикле for для печати содержимого справочника.

```
for last, first in directory:
print first, last, directory[last,first]
```

Цикл перебирает ключи справочника, которые являются кортежами. Происходит присваивание элементов каждого кортежа переменным last и first, затем выводится имя и соответствующий телефонный номер.

20.8. Последовательности: строки, списки, кортежи...

Мы уделяли внимание в основном спискам кортежей, но почти во всех примерах в этой главе используются и списки списков, кортежи кортежей, списки кортежей. Чтобы не перечислять все возможные комбинации, проще говорить о последовательностях последовательностей.

Во многих случаях различные виды последовательностей (строки, списки, кортежи) могут использоваться как взаимозаменяемые. Но как выбрать конкретный тип среди всех возможных? Начнем с очевидного: строки ограничены тем, что их элементами являются символы. К тому же строки неизменяемы. Если необходима возможность изменения символов в строке (а не просто создание новой строки), можно использовать вместо строк списки символов.

Списки более распространены, чем кортежи, в основном потому, что их можно изменять. Но ниже приведено несколько случаев, когда предпочтение отдается кортежам

1. В некоторых случаях, например, в операторе return, синтаксически проще создать кортеж, чем список. В других случаях можно предпочесть список.

2. Если мы хотим использовать последовательность как ключ в словаре, нам обязательно нужен неизменяемый тип, например, кортеж или строка.
3. Если последовательность используется в качестве аргумента функции, применение кортежей уменьшает вероятность незапланированного поведения благодаря ссылкам.

Так как кортежи неизменяемы, они не позволяют применять такие методы, как sort и reverse, которые изменяют содержимое списка. Однако Питон предоставляет встроенные функции sorted и reversed (сортировка и обращение), которые принимают любую последовательность в качестве параметра и возвращают новый список с теми же элементами в другом порядке.

20.9. Отладка

Списки, словари и кортежи являются примерами структур данных; в этой главе мы начали рассматривать сложные ("составные") структуры данных, например, списки кортежей, словари, которые содержат кортежи в качестве ключей и списки в качестве значений. Сложные структуры данных очень полезны, но они часто приводят к ошибкам, которые я называю ошибки формы; эти ошибки возникают, когда структура данных имеет неправильный тип, размер или состав, или, возможно, когда при написании некоторого кода вы забыли форму этих данных.

Например, если вы ожидаете список с одним целым числом, а вам предоставляется обычное целое число (не в списке), программа работать не будет.

При отладке программы и особенно при исправлении тяжелой ошибки пытайтесь выполнить четыре вещи.

Чтение: проверьте свой код, прочитайте его еще раз, чтобы убедиться, действительно ли он выражает то, что вы хотели.

Выполнение: экспериментируйте путем внесения изменений и запускайте различные версии. Часто, если вы вставили правильный код в правильное место в программе, проблема становится очевидной, но

иногда необходимо потратить значительное время на вспомогательную работу.

Размышляйте: найдите время на размышления! Что это за ошибка: синтаксическая, времени выполнения, семантическая? Какую информацию можно получить из сообщений об ошибках или из вывода программы? Какая ошибка может породить эту проблему? Какие последние изменения были внесены до появления ошибки?

Отступление: бывает так, что лучшее из того, что можно предпринять – это отменять последние изменения, пока вы не вернетесь к программе, которая работает. Затем можно начать пошаговое восстановления.

Начинающие программисты часто застревают на одном из этих действий и забывают о других.

Каждое действие имеет свои собственные ошибки. Например, чтение кода позволит обнаружить опечатку, но не поможет, если ошибка заключается в концептуальном непонимании. Если вы не понимаете, что делает ваша программа, вы не сможете обнаружить ошибку, даже если прочтете код 100 раз, потому что ошибка у вас в голове. Экспериментирование может помочь, особенно при запуске небольших простых тестов. Но если вы начнете экспериментировать без обдумывания и чтения кода, то в конце концов можете дойти до "программирования методом случайного блуждания", которое представляет собой процесс внесения случайных изменений до тех пор, пока программа не станет работать правильно. Излишне говорить, что случайное блуждание может длиться неопределенно долго. Не лучше ли немного подумать?

Отладка похожа на экспериментальную науку. Необходимо иметь хотя бы одну гипотезу о причинах проблемы. При наличии двух и более возможностей постарайтесь придумать тест, исключающий одну из них.

Небольшой перерыв помогает дальнейшим размышлениям. Также полезны обсуждения. Если вы объясните проблему кому-то (или даже себе), возможно, вы найдете ответ еще до того, как закончите задавать вопрос.

Однако даже лучшие способы отладки бессильны, когда в коде слишком

много ошибок или когда код, который вы хотите исправить, слишком большой и сложный. Иногда лучше отступить и упростить программу, пока не получится то, что работает и всё еще находится под вашим контролем.

Начинающие программисты зачастую неохотно вносят изменения, боясь удалить строку кода, даже когда она неправильная. Для спокойствия скопируйте программу в другой файл перед началом его правки. При необходимости легко можно будет вставить фрагменты исходного кода назад.

Поиск серьезных ошибок требует чтения, запусков, размышлений, а иногда и отмены последних изменений. Если вы застряли на одном из действий, попробуйте другие.

20.10. Глоссарий

Сравнимый (comparable): тип, два значения которого можно сравнивать друг с другом, определяя, что первое значение больше, меньше или равно второму. Сравнимые типы можно помещать в список и сортировать.

Структура данных (data structure): совокупность связанных значений, часто представленная в виде списков, словарей, кортежей и т.д.

DSU: Сокращение от "decorate-sort-undecorate," (декорирование-сортировка-раздекорирование) – метод, который включает в себя построение списка кортежей, сортировку и извлечение части результата.

Сборка (gather): операция формирования кортежа путем объединения произвольного числа аргументов.

Хешируемый (hashable): тип, имеющий хеш-функцию. Неизменяемые типы, такие как целые или вещественные числа и строки, являются хешируемыми; изменяемые, такие как списки и словари, – нет.

Разброс (scatter): операция использования последовательности в качестве списка аргументов.

Форма структуры данных (shape): совокупность типа, размера и состава

структуры данных.

Одноэлементный (singleton): список (или другая последовательность) с одним элементом.

Кортеж (tuple): неизменяемая последовательность элементов.

Присваивание кортежей (tuple assignment): присваивание последовательности, стоящей с правой стороны от оператора присваивания, кортежу переменных с левой стороны. Правая сторона сначала вычисляется, затем вычисленные значения присваиваются переменным с левой стороны.

20.11. Упражнения

Упражнение 20.1.

Доработайте первоначальную программу следующим образом: она должна читать и разбирать поле "From" каждого сообщения и извлекать адрес из него. С помощью словаря нужно для каждого человека подсчитать число отправленных им сообщений.

После завершения обработки данных выводится имя человека, отправившего наибольшее число сообщений, при помощи создания списка кортежей (количество, e-mail) из словаря и последующей сортировки списка в обратном порядке.

Пример:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Enter a file name: mbox-short.txt
cwen@iupui.edu 5
Enter a file name: mbox.txt
zqian@umich.edu 195
```

Упражнение 20.2.

Программа должна рассчитать распределение количества сообщений для каждого часа в сутках. Номер часа можно извлечь из поля "From"

сообщения, найдя в нем подстроку, задающую время, и разделив ее на части по двоеточиям. После подсчета количества сообщений для каждого часа выведите значения счетчиков, по одному на каждую строку, отсортировав их по номеру часа, как показано ниже.

Пример выполнения программы:

```
python timeofday.py
Enter a file name: mbox-short.txt

04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Упражнение 20.3.

Напишите функцию most_frequent (наиболее частые), которая получает строку и выводит ее буквы в порядке убывания частоты. Найдите образцы текста на разных языках и посмотрите, как частоты букв меняются в разных языках. Сравните свои результаты с таблицей по адресу ссылка: wikipedia.org/wiki/Letter_frequencies.

[1] Интересный факт: слово tuple — "кортеж" – произошло от названий последовательностей чисел различной длины: двухэлементных, трехэлементных, четырехэлементных, пяти-, шести- и т.д. (double, triple, quadruple, quituple, sextuple, …).

[2] Питон не воспринимает синтаксис буквально. Например, если попытаться выполнить команду для словаря, она не будет работать.

[3] Версия Питона 3.0 в данном случае работает немного иначе.

10 часто встречающихся слов

Видео

Регулярные выражения

Видео

Ранее нам часто приходилось, читая файлы, искать текстовые шаблоны и извлекать нужные нам фрагменты строк. Для этого мы применяли строковые методы, такие как `split` (расщепление) и `find` (поиск), и использовали списки и подстроки для выделения частей строк.

Задача поиска и выделения встречается очень часто, поэтому Питон содержит достаточно мощную библиотеку, работающую с регулярными выражениями, которая предоставляет элегантные решения большинства подобных задач. Причина, почему мы не рассматривали регулярные выражения раньше в этой книге, состоит в том, что это очень мощный инструмент, овладение которым не дается сразу, а к синтаксису регулярных выражений нужно привыкнуть.

Регулярные выражения представляют собой почти что небольшой самостоятельный язык программирования для поиска и разбора строк. Ему зачастую посвящают целые книги. В этой главе мы затронем лишь базовые возможности регулярных выражений. Более детальное описание доступно по адресам: ссылка: http://en.wikipedia.org/wiki/Regular_expression и ссылка: http://docs.python.org/library/re.html.

Библиотека для работы с регулярными выражениями должна быть импортирована в вашу программу. Простейший пример использования регулярных выражений – функция поиска `search()`. Следующая программа демонстрирует тривиальное использование функции поиска.

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
if re.search('From:', line) :
print line
```

Здесь открывается файл и в цикле просматриваются все его строки;

функция поиска регулярного выражения `search()` используется только для того, чтобы напечатать строки, содержащие фрагмент "From:". Данная программа не использует всей силы регулярных выражений, поскольку мы могли бы попросту воспользоваться строковым методом `line.find()` и получить тот же самый результат. Сила регулярных выражений раскрывается, когда мы добавляем специальные символы в шаблон поиска, что позволяет более точно задавать соответствие строк шаблону. Применение этих специальных символов в регулярном выражении дает возможность устанавливать сложное сопоставление и извлечение фрагментов текста с помощью совсем короткого программного кода.

Например, символ "^" (крышка) в регулярном выражении означает фрагмент в начале строки. Можно изменить нашу программу так, чтобы она находила только строки, содержащие слово "From: " в начале строки:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
if re.search('^From:', line) :
print line
```

Теперь шаблону сопоставляются только строки, начинающиеся с фрагмента "From: ". Это всё ещё совсем простой пример, который мы могли бы реализовать и с использованием метода `startswith()` из библиотеки работы со строками. Но он показывает, как специальные символы в регулярных выражениях расширяют наши возможности при сопоставлении фрагментов текста шаблону поиска.

23.1. Сопоставление символов в регулярных выражениях

Имеется ряд других специальных символов, дающих возможность строить еще более эффективные регулярные выражения. Наиболее часто используемый специальный символ – это точка, которая сопоставляется любому символу.

В следующем примере регулярное выражение "F..m:" сопоставляется любой из строк "From:", "Fxxm:", "F12m:", or "F!@m:", поскольку точка в регулярном выражении соответствует любому символу.

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
if re.search('^F..m:', line) :
print line
```

Это особенно эффективно в сочетании с возможностью указать, что символ может быть повторен произвольное количество раз, – для этого в регулярном выражении применяются специальные символы "*" или "+". Они означают, что вместо одного символа строки поиска сопоставляется ноль или более символов в случае звёздочки и один или более символов в случае знака плюс.

Мы можем еще более сузить множество строк, которые сопоставляются регулярному выражению, используя повторение произвольного символа (т.е. точки) в следующем примере:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
if re.search('^From:.+@', line) :
print line
```

Шаблон поиска "^From:.+@" успешно сопоставляется всем строкам, начинающимся со слова "From:", за которым следует один или более произвольных символов и затем символ @. Например, он соответствует следующей строке:

From: stephen.marquard@uct.ac.za

Часть шаблона ".+" можно представлять себе как фрагмент, который расширяется до соответствия всем символам между двоеточием и

символом @.

From:.+@

Удобно представлять плюс и звездочку как расширяющие символы. Например, в следующей строке символу @ шаблона будет соответствовать последний символ @ строки, тогда как фрагмент ".+" шаблона расширяется до всех символов после двоеточия, предшествующих ему:

From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen@iupui.edu

Можно заставить звездочку или плюс не быть столь "ненасытными", добавив другой специальный символ – см. подробную документацию по выключению их "жадного" поведения.

23.2. Извлечение данных с помощью регулярных выражений

Для выделения данных из строки Питон предоставляет метод `findall()`, извлекающий все подстроки, соответствующие регулярному выражению. Пусть мы хотим извлечь всё, что похоже на адрес электронной почты, из любой строки, независимо от ее формата. Например, мы хотим извлечь e-mail адрес из любой из следующих строк:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

Не хотелось бы писать отдельный код для каждого типа строк, разбирая по-своему каждую строку. Приведенная ниже программа использует метод `findall()` для нахождения всех строк с e-mail адресом и извлечения одного или нескольких адресов из таких строк.

```
import re
s = 'Hello from csev@umich.edu to cwen@iupui.edu about the meeting @2PM'
```

```
lst = re.findall('\S+@\S+', s)
print lst
```

Метод `findall()` производит поиск регулярного выражения в строке, переданной ему в качестве второго аргумента, и возвращает список всех ее подстрок, которые выглядят как e-mail адреса. Мы используем специальную двухсимвольную последовательность \S длины 2, которая сопоставляется любому непробельному (non-whitespace) символу. На выходе программы получим:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Сопоставляя регулярное выражение, мы находим все подстроки, имеющие хотя бы один непробельный символ, за которым следует символ @, после которого в свою очередь идет один или более непробельный символ. Шаблон "\S+" аналогично сопоставляется максимально длинной последовательности непробельных символов (это называют "жадным" поведением в регулярных выражениях).

Регулярное выражение сопоставляется дважды (csev@umich.edu и cwen@iupui.edu), но оно не соответствует подстроке "@2PM", поскольку перед символом @ нет непробельных символов. Мы можем использовать это регулярное выражение в программе, читающей все строки в файле и печатающей всё, что выглядит как e-mail адрес:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
x = re.findall('\S+@\S+', line)
if len(x) > 0 :
print x
```

Из каждой прочитанной строки извлекаются все подстроки, соответствующие нашему регулярному выражению. Поскольку метод `findall()` возвращает список, достаточно проверить, что число его элементов больше нуля, чтобы напечатать строки, содержащие e-mail адреса.

Применив программу к файлу `mbox.txt`, получим следующий результат:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Некоторые из этих e-mail адресов содержат некорректные символы "<" или ";" в начале либо конце. Укажем, что нас интересует лишь часть строки, начинающаяся и заканчивающаяся буквой или цифрой. Для этого используем другую возможность, предоставляемую регулярными выражениями. Квадратные скобки в них применяются для указания множества допустимых символов, используемых при сопоставлении. В этом смысле шаблон "\S" соответствует множеству всех непробельных символов. Теперь мы более явно укажем множество сопоставляемых символов.

Вот наше новое регулярное выражение:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

Оно несколько сложнее, теперь уже видно, почему регулярные выражения можно считать самостоятельным языком. Данное регулярное выражение означает, что мы ищем подстроки, начинающиеся с одиночной строчной буквы, прописной буквы или цифры "[a-zA-Z0-9]", за которой следует ноль или более непробельных символов "\S*", дальше идет символ @, потом ноль или более непробельных символов "\S*" и в конце строчная или прописная буква. Отметим, что мы заменили "+ " на "* " для указания нуля или более непробельных символов, поскольку фрагмент " [a-zA-Z0-9] " уже представляет собой один непробельный символ.

Помните, что "* " или "+ " применяются к одному символу, стоящему

непосредственно слева от звездочки или плюса. Если мы используем это выражение в нашей программе, выходные данные будут более чистыми:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
x = re.findall('[a-zA-Z0-9]\S+@\S+[a-zA-Z]', line)
if len(x) > 0 :
print x
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Отметим, что из строк, содержащих фрагмент "source@collab.sakaiproject.org", наше регулярное выражение исключило два символа в конце строки (">;"). Это произошло потому, что, добавив фрагмент "[a-zA-Z]" в конец нашего регулярного выражения, мы указали, что подстрока, которая сопоставляется регулярному выражению, обязательно должна заканчиваться буквой. Поэтому сопоставление заканчивается, как только мы доходим до символа ">" в конце "sakaiproject.org>;" – на последней "сопоставимой" букве (в данном случае "g"). Также отметим, что каждая строчка вывода программы представляет собой одноэлементный список Питона, единственным элементом которого является строка.

23.3. Сочетание поиска и извлечения

Пусть мы хотим найти числа во всех строках, которые начинаются с фрагмента "X-", например:

```
X-DSPAM-Confidence: 0.8475
```

```
X-DSPAM-Probability: 0.0000
```

Мы не хотим выделять числа в плавающем формате из любых строк – только из строк, имеющих вышеуказанный синтаксис. Для этого можно использовать следующее регулярное выражение:

```
^X-.*: [0-9.]+
```

Здесь указывается, что мы выбираем только строки, начинающиеся с фрагмента "X-", за которым следуют ноль или более произвольных символов ".*", затем двоеточие ":" и пробел. После пробела ищем один или более символов, каждый из которых является либо цифрой 0-9, либо точкой "[0-9.]+". Отметим, что точка внутри квадратных скобок обозначает обычный символ точки, а не произвольный символ (т.е. подстановка произвольного символа не действует внутри квадратных скобок).

Это очень компактное выражение, и оно выделяет из многообразия строк только те, которые нас интересуют:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
if re.search('^X\S*: [0-9.]+', line) :
print line
```

Выполнив программу, мы успешно отфильтруем только нужные нам строки.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

Но теперь необходимо решить проблему извлечения чисел с помощью

метода `split`. В принципе, и это совсем не сложно, но можно использовать другую возможность, предоставляемую регулярными выражениями – одновременный поиск и разбор.

Круглые скобки – это также специальные символы в регулярных выражениях. Когда мы ставим круглые скобки, они игнорируются при сопоставлении строки регулярному выражению, однако при использовании метода `findall()` скобки указывают, что, сопоставляя регулярное выражение целиком подстроке, мы извлекаем лишь ту часть подстроки, которая соответствует части регулярного выражения в скобках. Внесем следующее изменение в нашу программу:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
x = re.findall('^X\S*: ([0-9.]+)', line)
if len(x) > 0 :
print x
```

Вместо вызова метода `search()` мы заключаем в круглые скобки ту часть регулярного выражения, которая представляет число в плавающем формате; тем самым мы указываем, что метод `findall()` должен возвращать лишь часть сопоставленной подстроки, представляющую число.

На выходе этой программы получаем:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

Числа все еще содержатся в списке и должны быть преобразованы из строк в вещественные числа, но фактически мы уже воспользовались

мощью регулярных выражений, чтобы найти и извлечь информацию, которая нас интересует.

Еще один пример применения этой техники: рассмотрим файл, который содержит некоторое количество строк в форме:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Пусть нам нужно извлечь все номера ревизий (целые числа в концах подобных строк). Можно использовать следующую программу:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
x = re.findall('^Details:.*rev=([0-9.]+)', line)
if len(x) > 0:
print x
```

Здесь регулярное выражение означает, что мы ищем строки, начинающиеся со слова "Details:", за которым следует произвольное количество любых символов ".*", затем фрагмент "rev=" и далее одна или несколько цифр. Мы ищем строки, соответствующие всему выражению, но хотим извлечь из них только числа в концах строк, поэтому мы заключаем фрагмент "[0-9]+" в круглые скобки.

В результате работы программы получаем:

```
['39772']
['39771']
['39770']
['39769']
...
```

Помните, что фрагмент "[0-9]+" "жадный", он пытается выделить максимально большую подстроку из цифр перед ее извлечением. Это "жадное" поведение объясняет, почему выделяются все пять цифр каждого числа. Расширение происходит в обоих направлениях, пока не

встретится либо не-цифра, либо начало или конец строки.

Теперь мы можем использовать регулярные выражения, чтобы переписать упражнение, рассмотренное ранее в этой книге, в котором нас интересовало время суток каждого письма в электронной почте. Мы искали строки вида:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Мы хотим извлечь час из каждой такой строки. Раньше мы делали это с помощью двух вызовов метода `split`. Сначала строка разделялась на слова, затем извлекалось пятое слово, которое в свою очередь разделялось с помощью двоеточия для извлечения интересовавших нас двух символов.

Хотя это и работало, мы получили в результате довольно ненадежный, хрупкий (brittle) код, предполагающий, что исходные строки правильно отформатированы. Если добавить проверку ошибок (или большой блок try/except), чтобы программа не отказывала при некорректно отформатированных входных данных, код раздулся бы до 10-15 строк и стал бы трудно читаемым.

Ту же задачу можно решить намного проще с помощью следующего регулярного выражения:

```
^From .* [0-9][0-9]:
```

Оно означает, что мы ищем строки, начинающиеся с фрагмента "From " (обратите внимание на пробел!), за которым следует произвольное количество любых символов ".*", затем пробел и дальше две цифры "[0-9][0-9]", после которых стоит символ двоеточия. Это точное определение строк, которые мы ищем.

Для извлечения часа с помощью метода `findall()` мы заключаем в круглые скобки часть выражения, соответствующую двум цифрам:

```
^From .* ([0-9][0-9]):
```

В результате получаем программу:

```
import re
hand = open('mbox-short.txt')
for line in hand:
line = line.rstrip()
x = re.findall('^From .* ([0-9][0-9]):', line)
if len(x) > 0 : print x
```

На выходе она выдает

```
['09']
['18']
['16']
['15']
```

23.4. Символ "Escape"

Поскольку мы используем специальные символы в регулярных выражениях для указания начала или конца строки и подстановки произвольного символа, нам необходим также способ указать, что эти символы в отдельных случаях используются как "нормальные", чтобы просто сопоставить их с обычными символами, такими как доллар "$" или крышка "^".

Это можно сделать, поставив перед защищаемым символом обратную косую черту "\" (backslash). Например, можно найти обозначения денежных сумм с помощью следующего регулярного выражения:

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Поскольку мы поставили обратную косую черту перед знаком доллара, он будет сопоставляться с реальным символом доллара во входной строке, вместо того, чтобы обозначать конец входной строки; оставшаяся часть регулярного выражения сопоставляется с одной или более цифрой или точкой.

Замечание: символы внутри квадратных скобок не считаются специальными. Когда мы пишем "[0-9.]", это действительно означает

цифру или точку. Вне квадратных скобок точка означает подстановку произвольного символа. Внутри них точка означает точку.

23.5. Выводы

Пока мы лишь затронули тематику регулярных выражений, познакомившись с языком задания регулярных выражений. Они представляют собой шаблоны поиска, включающие специальные символы. С помощью регулярных выражений мы определяем задание для поисковой системы – что именно мы хотим найти и что извлечь из найденных строк. Ниже приведены некоторые специальные символы и специальные последовательности:

^

Соответствует началу строки.

$

Соответствует концу строки.

.

Соответствует любому символу (символ подстановки, wildcard).

\s

Соответствует пробельному символу (whitespace).

\S

Соответствует непробельному символу (противоположен \s).

*

Действует на непосредственно предшествующий символ и соответствует цепочке из нуля или более предшествующих символов.

*?

Действует на непосредственно предшествующий символ и соответствует цепочке из нуля или более предшествующих символов, выделенной в "нежадном" режиме.

```
+
```

Действует на непосредственно предшествующий символ и соответствует цепочке из одного или более предшествующих символов.

```
+?
```

Действует на непосредственно предшествующий символ и соответствует цепочке из одного или более предшествующих символов, выделенной в "нежадном" режиме.

```
[aeiou]
```

Соответствует одному символу из указанного набора. В данном примере может сопоставляться символам "a", "e", "i", "o", "u" и никаким другим.

```
[a-z0-9]
```

Можно указывать диапазон символов с помощью знака минус. В данном примере задается один символ, являющийся строчной буквой или цифрой.

```
[^A-Za-z]
```

Если первым символом в обозначении набора является крышка "^", то смысл обозначения меняется на обратный – любой символ, не входящий в указанный набор. В данном примере задается один символ, не являющийся прописной или строчной буквой.

```
( )
```

Круглые скобки в регулярном выражении игнорируются при сопоставлении строки и регулярного выражения, но позволяют извлечь указанную в скобках часть строки вместо всей строки при использовании метода `findall()`.

```
\b
```

Соответствует пустой строке в начале или конце слова.

\B

Соответствует пустой строке всюду, за исключением начала или конца слова.

\d

Соответствует десятичной цифре; эквивалентно выражению [0-9].

\D

Соответствует любому символу, отличному от десятичной цифры; эквивалентно выражению [^0-9].

23.6. Бонусный раздел для пользователей UNIX

Поиск в содержимом файлов с помощью регулярных выражений был встроен в операционную систему UNIX, начиная с 1960-х годов, и в настоящее время доступен практически во всех языках программированиях в той или иной форме.

Как правило, в системе UNIX имеется программа с консольным интерфейсом под названием grep (Generalized Regular Expression Parser), которая работает примерно так же, как описанный в этой главе метод `search()`. Таким образом, если вы работаете в системах Macintosh или Linux, вы можете попробовать следующую команду в консольном окне:

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

Команда указывает утилите `grep` напечатать все строки в файле `mbox-short.txt`, начинающиеся с фрагмента "From:".

Если вы немного поэкспериментируете с утилитой `grep` и прочитаете

документацию к ней, то найдете небольшие различия между регулярными выражениями Питона и регулярными выражениями, используемыми в `grep`. Например, `grep` не поддерживает непробельный символ "\S", так что придется использовать чуть более сложное обозначение "[^]", означающее попросту любой символ, отличный от пробела.

23.7. Отладка

Питон содержит простую встроенную документацию, которая пригодится, когда потребуется освежить знания и вспомнить правильное название того или иного метода. Документация доступна в интерпретаторе Питона в интерактивном режиме.

Воспользоваться интерактивной документацией можно с помощью команды `help()`.

```
>>> help()
Welcome to Python 2.6! This is the online help utility.
If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".
To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics". Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".
help> modules
```

Если вы знаете, какой модуль хотите использовать, можно воспользоваться командой `dir()` для перечисления его методов:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

Можно запросить короткую информацию по каждому конкретному методу:

```
>>> help (re.search)
Help on function search in module re:
search(pattern, string, flags=0)
Scan through string looking for a match to the pattern, returning
a match object, or None if no match was found.
>>>
```

Встроенная документация не слишком обширная, но может оказаться полезной, когда время ограничено или отсутствует доступ к веб-браузеру либо поисковой системе.

23.8. Глоссарий

Хрупкий код (brittle code): код, который работает, когда входные данные строго соответствуют заданному формату, но склонен к отказам, когда данные отклоняются от правильного формата. Мы называем такой код "хрупким", поскольку подобные программы легко ломаются.

Жадное сопоставление (greedy matching): сопоставление, при котором подстрока, задаваемая в регулярном выражении с помощью символов "+" и "*", расширяется до максимально возможного предела.

grep: команда, доступная в большинстве UNIX-систем, которая осуществляет поиск в содержимом текстовых файлов, выдавая строки, соответствующие регулярному выражению. Название команды является сокращением от "Generalized Regular Expression Parser".

Регулярное выражение: язык для задания сложных шаблонов поиска. Регулярные выражения могут содержать специальные символы, указывающие, что совпадение ищется в начале или в конце строки, и многие другие подобные возможности.

Символ подстановки (wild card): специальный символ, который при

сопоставлении соответствует любому символу. В регулярных выражениях в качестве символа подстановки используется точка.

23.9. Упражнения

Упражнение 23.1.

Напишите простую программу, моделирующую работу команды `grep` операционной системы Unix. Программа просит пользователя ввести регулярное выражение и затем подсчитывает количество строк, соответствующих этому выражению:

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author
$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-
$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

Упражнение 11.2.

Напишите программу, которая отыскивает в заданном файле все строки вида `New Revision: 39772`. Из каждой такой строки извлекается число с помощью регулярного выражения и метода `findall()`. Программа должна вычислить и напечатать среднее арифметическое всех этих чисел.

```
Enter file: mbox.txt
38549.7949721
Enter file: mbox-short.txt
39756.9259259
```

Сетевые программы

Видео

Хотя большая часть примеров в этой книге до сих пор фокусировалась на чтении файлов и поиске данных в них, есть много других источников информации, и важнейший из них – Интернет.

В этой главе мы попробуем исполнить роль веб-браузера и получить веб-страницу, используя протокол передачи гипертекста HTTP (HyperText Transport Protocol). Затем мы прочитаем содержимое страницы и осуществим его разбор.

23.1. Протокол передачи гипертекста – HTTP

Сетевой протокол, на котором основана вся мощь Интернет, на самом деле совсем простой, и его поддержка встроена в Питон – это так называемый механизм сокетов, пользуясь которым, совсем несложно устанавливать сетевые соединения и получать данные из сети в программах Питона.

Сокет в основном аналогичен файлу, за тем исключением, что он обеспечивает двустороннее соединение между двумя программами.

Вы можете как читать, так и записывать данные по одному и тому же сокету. Если вы что-то пишете в сокет, то эти данные передаются по сети парному приложению, работающему на другом конце сокета. Если вы читаете из сокета, то получаете данные, которые парное приложение послало вам. Но если вы пытаетесь прочесть данные, когда парное приложение на другом конце сокета еще ничего не послало – вы попросту ничего не делаете и ждёте. Если приложения на обоих концах сокета ждут прихода данных, не посылая ничего, то они будут ждать очень долго.

Поэтому для программ, взаимодействующих через Интернет, очень важно придерживаться некоторого протокола. Протокол – это набор четких правил, которые определяют, кто должен начинать обмен, какими должны быть ответы на полученные сообщения, кто должен посылать следующее сообщение и т.п. В некотором смысле два

приложения на концах сокета совместно исполняют танец и не должны наступать на ноги друг другу.

Существует множество документов, описывающих подобные сетевые протоколы. Протокол передачи гипертекста приводится в следующем документе: ссылка: http://www.w3.org/Protocols/rfc2616/rfc2616.txt.

Это длинный и сложный 176-страничный документ с множеством деталей. Если вам интересно, можете прочитать его. Но если прочитать лишь страницу 36 стандарта RFC2616, вы найдете описание синтаксиса запроса GET. Более точно, вы прочитаете, что для получения документа от веб-сервера нужно установить соединение с сервером по адресу www.py4inf.com и порту 80 и передать ему текстовую строку вида:

```
GET http://www.py4inf.com/code/romeo.txt
HTTP/1.0
```

Здесь второй параметр – это адрес запрашиваемой веб-страницы; затем нужно послать также пустую строку. Веб-сервер отвечает на запрос, посылая некоторую служебную ("заголовочную") информацию о запрашиваемом документе, пустую строку и затем содержимое документа.

23.2. Самый простой в мире веб-браузер

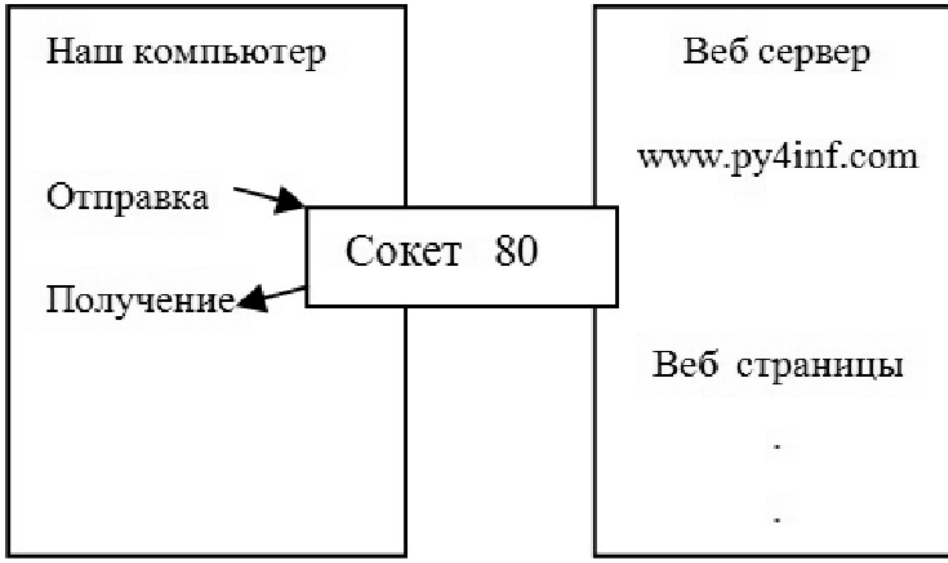
Наверное, самый простой способ продемонстрировать работу протокола HTTP – это написать программу, которая устанавливает соединение с веб-сервером, запрашивает документ в соответствии с протоколом HTTP и печатает то, что в ответ присылает сервер.

```
import socket
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('www.py4inf.com', 80))
mysock.send('GET http://www.py4inf.com/code/romeo.txt HTTP/1.0\n\n')

while True:
data = mysock.recv(512)
if ( len(data) < 1 ) :
```

```
break
print data
mysock.close()
```

Сначала программа устанавливает соединение по порту 80 с сервером на узле www.py4inf.com. Поскольку наша программа исполняет роль "веб-браузера", протокол HTTP предписывает послать команду GET и затем пустую строку.



После посылки пустой строки следует цикл, который принимает данные из сокета порциями по 512 символов и печатает их до тех пор, пока не останется непрочитанных данных (т.е. функция `recv()` не вернёт пустую строку).

На выходе программы получаем:

```
HTTP/1.1 200 OK
Date: Sun, 14 Mar 2010 23:52:41 GMT
Server: Apache
Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT
ETag: "143c1b33-a7-4b395bea"
Accept-Ranges: bytes
```

```
Content-Length: 167
Connection: close
Content-Type: text/plain
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Вывод программы начинается с заголовочной информации, которую сервер посылает для описания документа. Например, заголовочная строка Content-Type указывает, что это простой текстовый документ (text/plain). После того, как сервер пересылает нам всю заголовочную информацию, он посылает пустую строку, отделяющую служебное описание документа от его содержимого, и дальше пересылает текст, содержащийся в файле `romeo.txt`.

Этот пример показывает, как устанавливается низкоуровневое соединение с помощью сокетов. Сокеты можно использовать для обмена с веб-сервером, сервером электронной почты и многими другими типами серверов. Всё, что требуется – это разыскать документ, который описывает протокол обмена, и записать программный код, пересылающий и принимающий данные в соответствии с этим протоколом. Однако, поскольку самый часто используемый протокол – это HTTP (протокол Всемирной паутины), Питон имеет специальную библиотеку, обеспечивающую поддержку протокола HTTP для получения документов и данных через Интернет.

23.3. Получение веб-страниц с использованием библиотеки urllib

Библиотека `urllib` максимально упрощает получение веб-страниц и обработку их содержимого в программах Питона. Используя `urllib`, мы работаем с веб-страницами почти так же, как с файлами. Нужно всего лишь указать, какую веб-страницу мы хотим получить, дальше уже сама библиотека `urllib` отрабатывает все детали протокола HTTP.

Эквивалентный код для чтения файла `romeo.txt` из сети с помощью

`urllib` записывается следующим образом:

```
import urllib
fhand = urllib.urlopen('http://www.py4inf.com/code/romeo.txt')
for line in fhand:
print line.strip()
```

После того, как веб-страница открыта с помощью метода `urllib.urlopen`, можно работать с ней как с файлом, читая ее содержимое в цикле `for`. На выходе программа выдает только содержимое файла. Заголовочная информация также пересылается, но код библиотеки `urllib` проглатывает заголовки и возвращает нам только данные:

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Для примера можно написать программу, получающую содержимое файла `romeo.txt` и подсчитывающую частоту каждого слова в тексте:

```
import urllib
counts = dict()
fhand = urllib.urlopen('http://www.py4inf.com/code/romeo.txt')
for line in fhand:
words = line.split()
for word in words:
counts[word] = counts.get(word,0) + 1
print counts
```

Еще раз: как только мы открыли веб-страницу, можно читать ее точно так же, как и локальный файл на компьютере.

23.4. Разбор HTML и извлечение информации из веб-страниц (Web scraping)

Очень часто возможности библиотеки `urllib` Питона используются при извлечении информации во всемирной паутине (Web scraping). Мы пишем программу, которая притворяется веб-браузером, получает Интернет-страницы и затем анализирует их содержимое, находя в них нужные текстовые шаблоны.

Например, поисковая машина, подобная Google, начинает свою работу с какой-нибудь веб-страницы, извлекает из нее ссылки на другие страницы, получает по сети содержимое этих страниц, извлекает из них ссылки и так далее. Используя подобную технику, пауки Google прокладывают свой путь практически через все страницы в сети. Google принимает частоту найденных подобным образом ссылок на конкретную веб-страницу как меру "важности" этой страницы, определяющую, насколько высоко она будет помещена в ответах на поисковые запросы.

23.5. Разбор HTML-страниц с помощью регулярных выражений

Один из самых простых способов анализа HTML-страниц – использование регулярных выражений для поиска и извлечения подстрок, соответствующих определенным шаблонам. Рассмотрим простейшую веб-страницу:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

Можно построить регулярное выражение для нахождения и извлечения ссылок из приведенного выше текста:

```
href="http://.+?"
```

Это регулярное выражение соответсвует подстрокам, начинающимся с фрагмента "href="http://", за которым следует один или несколько произвольных символов ".+?" и далее закрывающая двойная кавычка.

Вопросительный знак после плюса в ".+?" указывает, что сопоставление подстроки шаблону должно происходить в "нежадном" режиме вместо используемого по умолчанию "жадного". При "нежадном" сопоставлении мы пытаемся найти максимально короткую строку, соответствующую шаблону, при "жадном" – максимально длинную. Нужно еще добавить круглые скобки в наше регулярное выражение, чтобы указать, какую часть сопоставленной шаблону строки мы хотим извлечь. В результате получим следующую программу:

```
import urllib
import re
url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
links = re.findall('href="(http://.*?)"', html)
for link in links:
print link
```

Метод `findall` из библиотеки работы с регулярными выражениями возвращает нам список всех подстрок, соответствующих нашему регулярному выражению, причем для каждой такой подстроки выдается только ее часть, заключенная между двумя двойными кавычками.

В результате работы программы получим:

```
python urlregex.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
python urlregex.py
Enter - http://www.py4inf.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.py4inf.com/code
http://www.lib.umich.edu/espresso-book-machine
http:// www.py4inf.com/py4inf-slides.zip
```

Регулярные выражения прекрасно работают, когда HTML-текст правильно и предсказуемо отформатирован. Но, поскольку в сети огромное количество не вполне корректных страниц (формат HTML не является строгим), программа, основанная лишь на регулярных

выражениях, либо пропускает некоторые допустимые ссылки, либо выдает неправильные данные. Эта проблема может быть решена путем использования библиотеки для надежного разбора HTML.

23.6. Разбор HTML-страниц с помощью библиотеки BeautifulSoup

В Питоне есть несколько библиотек, помогающих при разборе HTML-текста и извлечении данных из веб-страниц. Каждая из этих библиотек имеет свои преимущества и недостатки.

Вы можете выбрать одну из них, основываясь на своих потребностях. Например, задача разбора HTML-текста и извлечения ссылок из него легко решается с помощью библиотеки BeautifulSoup. Вы можете скачать и установить эту библиотеку с адреса ссылка: www.crummy.com. Можно даже не устанавливая библиотеку, просто поместить файл BeautifulSoup.py в тот же каталог, что и ваше приложение.

Несмотря на то, что HTML-документ выглядит как XML и некоторые страницы тщательно сконструированы так, чтобы удовлетворять строгим правилам XML, большая часть HTML-страниц сформирована неправильно в том смысле, что XML-парсер отвергает подобные страницы целиком как некорректные. Но библиотека `BeautifulSoup` терпимо относится даже к очень неряшливым страницам и позволяет извлекать из них нужную информацию.

Мы будем использовать библиотеку `urllib`, чтобы читать веб-страницы, и затем библиотеку `BeautifulSoup`, чтобы извлекать тексты ссылок (т.е. аттрибут href) из тегов <a> (anchor).

```
import urllib
from BeautifulSoup import *
url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
soup = BeautifulSoup(html)

# Retrieve all of the anchor tags
tags = soup('a')
```

```
for tag in tags:
print tag.get('href', None)
```

Программа предлагает ввести веб-адрес, открывает веб-страницу, читает ее содержимое и передает данные парсеру `BeautifulSoup`, а затем извлекает все теги <a> и выводит атрибут `href` (т.е. гипертекстовую ссылку) для каждого тега.

После запуска программы на выходе получим:

```
python urllinks.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm
python urllinks.py
Enter - http://www.py4inf.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.si502.com/
http://www.lib.umich.edu/espresso-book-machine
http://www.py4inf.com/code
http://www.pythonlearn.com/
```

Можно использовать `BeautifulSoup` для извлечения различных частей каждого тега, как в следующей программе:

```
import urllib
from BeautifulSoup import *
url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
soup = BeautifulSoup(html)
# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
# Look at the parts of a tag
print 'TAG:',tag
print 'URL:',tag.get('href', None)
print 'Content:',tag.contents[0]
```

```
print 'Attrs:',tag.attrs
```

В результате получаем:

```
python urllink2.py

Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: [u'\nSecond Page']
Attrs: [(u'href', u'http://www.dr-chuck.com/page2.htm')]
```

Эти примеры лишь отчасти демонстрируют силу библиотеки `BeautifulSoup` в применении к разбору HTML-гипертекста. Для более детального знакомства см. документацию и примеры по адресу ссылка: www.crummy.com.

23.7. Чтение бинарных файлов с помощью urllib

Часто нужно получить из сети нетекстовый (бинарный) файл, например, изображение или видео-файл. Данные в таких файлах обычно не используются для печати, но с помощью `urllib` нетрудно просто скопировать содержимое сетевого файла на жесткий диск.

Открываем URL и используем метод `read` для скачивания всего содержимого документа в строковую переменную `img`, затем записываем эту информацию в локальный файл:

```
img = urllib.urlopen('http://www.py4inf.com/cover.jpg').read()
fhand = open('cover.jpg', 'w')
fhand.write(img)
fhand.close()
```

Эта программа читает по сети все данные целиком и сохраняет их в переменной `img`, содержащейся в оперативной памяти компьютера,

затем открывает файл `cover.jpg` и записывает данные на диск.

Это работает, когда размер сетевого файла меньше, чем объем памяти вашего компьютера. Однако при чтении огромного аудио- или видеофайла программа может привести к отказу или в лучшем случае начнет работать крайне медленно, когда физическая память компьютера будет исчерпана.

Чтобы избежать подобной ситуации, мы читаем данные из сети блоками и записываем каждый блок на диск перед тем, как считать следующий блок. При подобном способе работы программа может прочитать сетевой файл любого размера и при этом избежать переполнения памяти компьютера.

```
import urllib
img = urllib.urlopen('http://www.py4inf.com/cover.jpg')
fhand = open('cover.jpg', 'w')
size = 0
while True:
info = img.read(100000)
if len(info) < 1 : break
size = size + len(info)
fhand.write(info)
print size,'characters copied.'
fhand.close()
```

В этом примере мы читаем за один раз блок размеров в 100000 байтов и записываем его файл `cover.jpg` перед чтением следующей порции данных из сети. После запуска программа выдает:

```
python curl2.py
568248 characters copied.
```

Если у вас Unix- или Macintosh-компьютер, то, скорее всего, в операционной системе есть команда, которая выполняет описанную выше операцию:

```
curl -O http://www.py4inf.com/cover.jpg
```

Название команды `curl` является сокращением от "copy URL". Поэтому файлы с программами Питона из последних двух примеров названы `curl1.py` и `curl2.py` – они делают то же самое, что и команда `curl`. Также имеется программа `curl3.py`, которая решает ту же задачу чуть более эффективно – ее тоже можно использовать в качестве образца при написании собственных программ.

23.8. Глоссарий

BeautifulSoup[1] : библиотека Питона для разбора HTML-документов и извлечения данных из них, которая, подобно большинству браузеров, принимает даже плохо сформированные HTML-документы. Код библиотеки `BeautifulSoup` можно скачать по адресу ссылка: www.crummy.com

Порт: число, которое обычно определяет, с каким именно приложением вы общаетесь, когда устанавливается соединение с сервером через сокет. Например, передача данных во всемирной паутине обычно использует порт 80, электронная почта – порт 25.

Scrape – извлечение, дословно "выскабливание" данных: процесс, при котором сетевая программа, притворяясь веб-браузером, получает по сети веб-страницы и затем анализирует их содержимое. Часто подобные программы следуют по ссылкам, содержащимся в прочитанной странице, чтобы перейти к следующей странице и таким образом пройти всю сеть, например, социальную.

Сокет: сетевое соединение между двумя приложениями, при котором эти приложения могут посылать и принимать данные в обоих направлениях.

Паук (spider): способ работы сетевой поисковой машины, которая считывает страницу, затем все страницы, на которые она ссылается, и так далее, пока в конце-концов не будут пройдены почти все страницы в Интернет. Используется при построении поисковых систем.

23.9. Упражнения

Упражнение 23.1.

Измените использующую механизм сокетов программу `socket1.py` так, чтобы она сначала запрашивала у пользователя адрес веб-страницы (URL) и затем считывала её. Можно использовать разделитель '/' и строковый метод `split` для того, чтобы разбить URL на компоненты и извлечь адрес сетевого узла, который необходим при установке соединения через сокет. Добавьте проверку ошибок, используя конструкцию `try-except`, для обработки ситуации, когда пользователь ввел неправильно сформированный или несуществующий URL.

Упражнение 23.2.

Измените программу, использующую механизм сокетов, так, чтобы она подсчитывала количество полученных символов и прекращала печатать текст после того, как напечатано 3000 символов. Однако читать документ программа должна до конца – для подсчета числа символов во всём документе. По окончании чтения число символов должно быть напечатано.

Упражнение 23.3.

Используйте `urllib` для того, чтобы иным способом решить предыдущее упражнение:

1. получить документ из сети,
2. напечатать не более чем 3000 его начальных символов,
3. подсчитать общее число символов в документе.

В данном упражнении можно не беспокоиться о заголовочной информации к документу, печатаются лишь первые 3000 символов его содержимого.

Упражнение 23.4.

Измените программу `urllinks.py` так, чтобы извлечь теги <p> ("paragraph" – абзац) из полученного HTML-документа, подсчитать их число и на выходе напечатать количество абзацев в документе. Не

нужно печатать текст абзацев – только подсчитать их. Протестируйте вашу программу на нескольких небольших веб-страницах и затем попробуйте на каких-нибудь объемных страницах.

Упражнение 23.5 (более сложное).

Измените программу, использующую сокеты, так, чтобы она печатала только данные после того, как получены заголовочная информация и пустая строка. Помните, что метод `recv` получает символы (в том числе символ перехода на новую строку), а не строки.

[1] Название происходит от распространенного выражения "tag soup" – суп, смесь из тегов, которое употребляется для плохо сформированных HTML-документов. Библиотека BeautifulSoup – "Прекрасный суп", по-видимому, хорошо разбирается в подобном супе. – прим. перев.

Поиск тегов

Видео

Использование Веб-служб

Видео

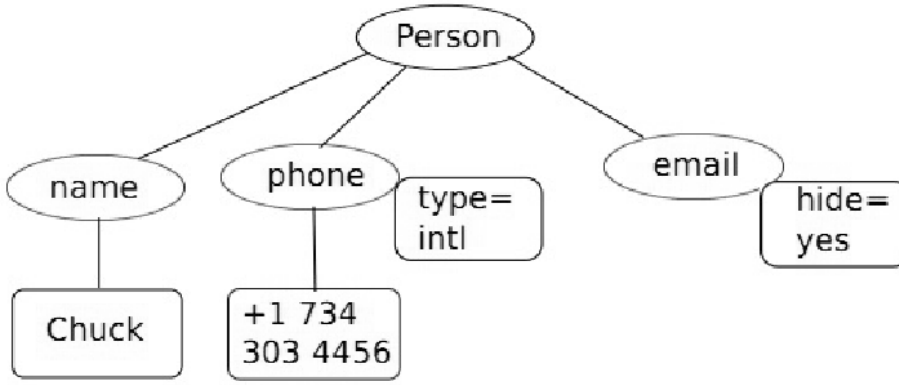
Поскольку несложно получать HTTP-документы по сети и осуществлять их разбор с помощью специальных программ, естественно применить подход, при котором создаются документы, предназначенные для других программ (не имеется в виду HTML, который показывается браузером). Чаще всего, когда две сетевые программы обмениваются данными через сеть, они используют формат данных, который называется "Расширяемым языком разметки" или XML (eXtensible Markup Language).

26.1. Расширяемый язык разметки – XML

XML похож на HTML, но более четко структурирован. Вот пример XML-документа:

```
<person>
<name>Chuck</name>
<phone type="intl">
+1 734 303 4456
</phone>
<email hide="yes"/>
</person>
```

Часто полезно представлять XML-документ как имеющий структуру дерева, например, в данном случае тег верхнего уровня (корневой тег) – это person, остальные теги, такие, как phone, изображаются как дети своих родительских узлов.



26.2. Разбор XML

Ниже приведена несложная программа, которая осуществляет разбор XML-документа и извлекает из него некоторые элементы:

```
import xml.etree.ElementTree as ET
data = '''
<person>
<name>Chuck</name>
<phone type="intl">
+1 734 303 4456
</phone>
<email hide="yes"/>
</person>'''
tree = ET.fromstring(data)
print 'Name:',tree.find('name').text
print 'Attr:',tree.find('email').get('hide')
```

Метод `fromstring` преобразует строковое представление XML-документа в дерево XML-узлов. Когда XML-документ уже представлен в виде дерева, есть целая серия методов для извлечения порций данных из него.

Функция `find` просматривает дерево XML и извлекает узел, который соответствует указанному тегу. Каждый узел может иметь некоторый

текст, а также атрибуты (например, аттрибут "hide" – "скрыть") и "детские" узлы.

Каждый узел можно рассматривать как корень поддерева, выходящего из него.

```
Name: Chuck
Attr: yes
```

Использование XML-парсера (т.е. программы, осуществляющей разбор и синтаксический анализ XML-текста), такого, как `ElementTree`, имеет то преимущество, что, несмотря на простоту рассмотренного примера, синтаксис XML подчиняется множеству правил, и использование `ElementTree` позволяет нам извлекать данные из XML-документа, не тратя время на изучение синтаксиса XML.

26.3. Циклы по узлам

Часто XML имеет многочисленные однотипные узлы, и нам приходится писать циклы, чтобы обрабатывать их. В следующем примере в цикле перебираются все узлы, соответствующие тегу `user`:

```
import xml.etree.ElementTree as ET
input = '''
<stuff>
<users>
<user x="2">
<id>001</id>
<name>Chuck</name>
</user>
<user x="7">
<id>009</id>
<name>Brent</name>
</user>
</users>
</stuff>'''
stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
```

```
print 'User count:', len(lst)
for item in lst:
print 'Name', item.find('name').text
print 'Id', item.find('id').text
print 'Attribute', item.get('x')
```

Метод `findall` извлекает из документа, представленного в виде XML-дерева, список поддеревьев, корни которых являются узлами, соответствующими тегу user. Затем мы используем цикл `for`, который для каждого узла user печатает имя пользователя (имя содержится в узле name, который является непосредственным потомком узла user) и идентификатор (узел id), а также значение атрибута x узла user.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

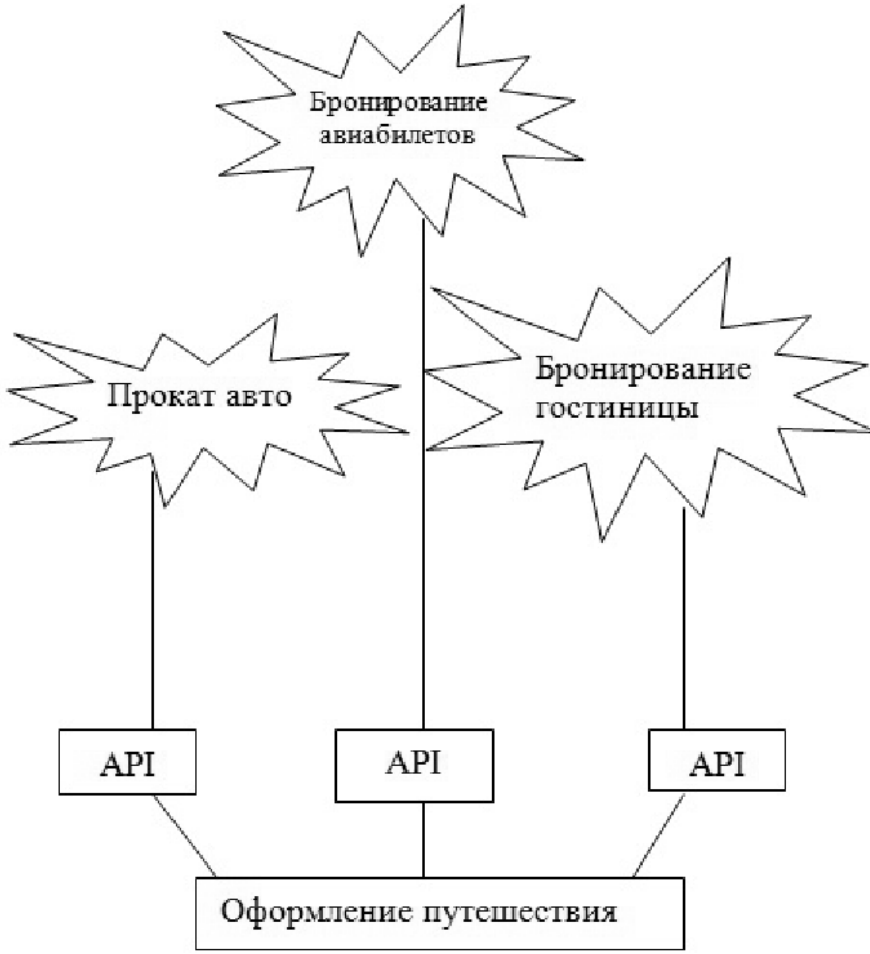
26.4. Интерфейсы прикладного программирования (API – Application Programming Interface)

Протокол передачи гипертекста HTTP делает возможным обмен данными между приложениями, а язык XML дает способ представления сложных данных, пересылаемых приложениями. Следующим шагом является определение и документирование "контрактов", заключаемых между приложениями. Общее название для подобных контрактов – Интерфейсы прикладного программирования (Application Programming Interfaces), или просто API.

Когда мы используем API, то в общем случае одна программа предоставляет набор сервисов, доступных другим программам для использования, и публикует API (т.е. правила), которые нужно соблюдать, чтобы воспользоваться этими сервисами. Подход, когда мы

разрабатываем программы, функционирование которых включает доступ к сервисам, предоставляемым другими программами, называется Сервисно-ориентированной архитектурой (Service-Oriented Architecture) или SOA.

При использовании SOA приложение при своей работе пользуется сервисами других приложений. Без использования SOA приложение представляет собой отдельную (stand-alone) программу, содержащую внутри себя весь код, который необходим для ее работы. Мы встречаем множество примеров SOA, когда используем сеть. Можно войти на единый веб-сайт и заказать там авиабилеты, забронировать отель и арендовать автомобиль. Данные по отелям не хранятся на компьютерах, отвечающих за авиаперевозки. Вместо этого компьютеры авиалиний связываются с компьютерами отелей, получают от них необходимые данные и представляют их пользователю. Если пользователь согласен сделать заказ отеля через сайт авиалиний, последний использует другой сетевой сервис, предоставляемый системой, отвечающей за отели, чтобы выполнить реальный заказ отеля. И в тот момент, когда с вашей кредитной карты снимаются деньги за всю транзакцию целиком, другие компьютеры также вовлечены в этот процесс.



Сервис-ориентированная архитектура имеет множество преимуществ, включая следующие: (1) мы всегда храним лишь одну копию данных – это особенно важно при совершении таких действий, как бронирование отеля, когда важно не сделать заказ дважды; (2) собственники данных могут установить правила их использования. При этом SOA-системы должны быть тщательно разработаны, чтобы обеспечивать хорошую производительность и удовлетворять потребностям пользователей.

Когда приложение предоставляет через сеть набор услуг согласно своему API, мы называем это веб-службой.

26.5. Веб-службы Твиттера

Возможно, вы знакомы с сайтом Twitter и его приложениями ссылка: http://www.twitter.com.

У Твиттера есть уникальный подход к его API/веб-службам, в которых все данные доступны приложениям, не относящимся к Твиттеру, с помощью Твиттер-API.

Так как Твиттер очень либерален в отношении доступа к своим данным, он позволил тысячам разработчиков программного обеспечения создавать собственные приложения, основанные на программном обеспечении Твиттера. Эти дополнительные приложения увеличивают значение Твиттера, делая его намного большим, чем просто веб-сайт. Веб-службы Твиттера позволяют создавать новые приложения, о которых команда Твиттера даже и не задумывалась. По статистике, более 90 процентов обращений к Твиттеру происходит через API (т.е. не через веб-интерфейс сайта ссылка: twitter.com). Документацию API Твиттера можно просмотреть по ссылке: ссылка: http://apiwiki.twitter.com/.

API Твиттера является примером типа REST-стиля[1)] организации веб-служб. Например, используем Твиттер-API для извлечения списка пользователей-друзей и их статусов. Чтобы посмотреть список друзей пользователя drchuck, перейдем по ссылке ссылка: http://api.twitter.com/1/statuses/friends/drchuck.xml

Не всякий браузер корректно отображает XML. Однако всегда можно увидеть возвращаемый Твиттером XML-текст, посмотрев исходный код полученной "веб-страницы".

Получить этот XML-код можно и с помощью Питона, используя библиотеку `urllib`:

```
import urllib
TWITTER_URL = 'http://api.twitter.com/l/statuses/friends/ACCT.xml'
while True:
print ''
acct = raw_input('Enter Twitter Account:')
if ( len(acct) < 1 ) : break
```

```
url = TWITTER_URL.replace('ACCT', acct)
print 'Retrieving', url
document = urllib.urlopen (url).read()
print document[:250]
```

Программа запрашивает название учетной записи Твиттера и, используя Твиттер-API, открывает URL, содержащую список друзей и их статус, получает текст URL и печатает первые 250 символов текста.

```
python twitter1.py

Enter Twitter Account:drchuck
Retrieving http://api.twitter.com/l/statuses/friends/drchuck.xml
<?xml version="1.0" encoding="UTF-8"?>
<users type="array">
<user>
<id>115636613</id>
<name>Steve Coppin</name>
<screen_name>steve_coppin</screen_name>
<location>Kent, UK</location>
<description>Software developing, best practicing, agile e
Enter Twitter Account:
```

В этом приложении мы получаем из сети XML-код точно так же, как и HTML-страницу. Если необходимо извлечь данные из XML, можно было бы использовать строковые функции Питона, но это непросто, поскольку требует погружения в детали XML.

Извлеченный XML-код может выглядеть примерно так:

```
<?xml version="1.0" encoding="UTF-8"?>
<users type="array">
<user>
<id>115636613</id>
<name>Steve Coppin</name>
<screen_name>steve_coppin</screen_name>
<location>Kent, UK</location>
<status>
```

```
<id>10174607039</id>
<source>web</source>
</status>
</user>
<user>
<id>17428929</id>
<name>davidkocher</name>
<screen_name>davidkocher</screen_name>
<location>Bern</location>
<status>
<id>10306231257</id>
<text>@MikeGrace If possible please post a detailed bug report </text>
</status>
</user>
...
```

Верхний тег – "users", он включает в себя несколько тегов "user", которые являются его потомками. Тег "status", в свою очередь, является потомком каждого тега "user".

26.6. Обработка XML-данных, полученных с помощью API

После получения правильно структурированных XML-данных с помощью API мы обычно используем XML-парсер, такой, как `ElementTree`, для извлечения информации из XML. В приведенной ниже программе мы получаем список друзей и их статусы, пользуясь API Твиттера, и затем разбираем возвращенный XML-код, чтобы напечатать первых четырех друзей и их статус.

```
import urllib
import xml.etree.ElementTree as ET
TWITTER_URL = 'http://api.twitter.com/1/statuses/friends/ACCT.xml'
while True:
print ''
acct = raw_input('Enter Twitter Account:')
if ( len(acct) < 1 ) : break
```

```
url = TWITTER_URL.replace('ACCT', acct)
print 'Retrieving', url
document = urllib.urlopen (url).read()
print 'Retrieved', len(document), 'characters.'
tree = ET.fromstring(document)
count = 0
for user in tree.findall('user'):
count = count + 1
if count > 4 : break
print user.find('screen_name').text
status = user.find('status')
if status :
txt = status.find('text').text
print ' ',txt[:50]
```

С помощью метода `findall` мы получаем список узлов с тегом user и затем перебираем элементы списка в цикле `for`. Для каждого узла user мы извлекаем и печатаем текст сыновнего узла screen_name и затем извлекаем сыновний узел status. Если последний существует, то мы печатаем первые 50 символов содержащегося в нем текста.

Идея программы проста и понятна, мы используем методы `findall` и `find` для извлечения либо списка узлов, либо одного узла и затем в случае, когда узел представляет сложный элемент с множеством подчиненных узлов, мы погружаемся глубже по дереву до тех пор, пока не находим интересующий нас текстовый элемент.

Выполнив программу, получим:

```
python twitter2.py
Enter Twitter Account:drchuck
Retrieving http://api.twitter.com/l/statuses/friends/drchuck.xml

Retrieved 193310 characters.
steve_coppin
Looking forward to some "oh no the markets closed,
davidkocher
@MikeGrace If possible please post a detailed bug
```

```
hrheingold
From today's Columbia Journalism Review, on crap d
huge_idea
@drchuck #cnx2010 misses you, too. Thanks for co
Enter Twitter Account:hrheingold
Retrieving http://api.twitter.com/l/statuses/friends/hrheingold.xml
Retrieved 208081 characters.
carr2n
RT @tysone: Saturday's proclaimation by @carr2n pr
tiffanyshlain
RT @ScottKirsner: Turning smartphones into a tool
soniasimone
@ACCompanyC Funny, smart, cute, and also nice! He
JenStone7617
Watching "Changing The Equation: High Tech Answers
Enter Twitter Account:
```

Код для разбора XML и извлечения нужных полей с помощью библиотеки `ElementTree` составляет всего несколько строк, это намного проще, чем использование строковых методов Питона для решения аналогичной задачи.

26.7. Глоссарий

API: Интерфейс прикладного программирования (Application Program Interface) – соглашение между приложениями, которое устанавливает правила взаимодействия между двумя компонентами приложений.

ElementTree: встроенная библиотека Питон, используемая для разбора XML-данных.

XML: Расширяемый язык разметки (eXtensible Markup Language) – формат для представления структурированных данных.

REST: сокращение от REpresentational State Transfer (передача представления состояния) – стиль построения сетевых служб, обеспечивающий доступ к сетевым ресурсам с помощью протокола HTTP. С его помощью по сети передается документ, представляющий

текущее состояние ресурса.

SOA: Сервис-ориентированная архитектура (Service Oriented Architecture) – когда приложение состоит из компонентов взаимодействующих через сеть.

26.8. Упражнения

Упражнение 26.1.

Измените программу, получающую данные из Твиттера (twitter2.py), так, чтобы она дополнительно печатала местонахождение каждого из друзей ниже его имени, отступив на 2 пробела от начала строки:

```
Enter Twitter Account:drchuck
Retrieving http://api.twitter.com/l/statuses/friends/drchuck.xml
Retrieved 194533 characters.
steve_coppin
  Kent, UK
Looking forward to some "oh no the markets closed,
davidkocher
  Bern
@MikeGrace If possible please post a detailed bug
hrheingold
  San Francisco Bay Area
RT @barrywellman: Lovely AmBerhSci Internet & Comm
huge_idea
  Boston, MA
@drchuck #cnx2010 misses you, too. Thanks for co
```

[1] REST – сокращение от Representational State Transfer – стиль построения сетевых систем данных, при которых данные, представляющие текущее состояние некоторого ресурса, передаются от сервера клиентам по их запросам. – прим. перев.

Использование баз данных и языка структурированных запросов (SQL)

26.1. Что такое база данных?

Презентация для лекции 26.ppt скачать: http://old.intuit.ru/department/pl/pythoninf/26/26.ppt.

База данных – это файл, используемый специально для хранения данных. Большая часть баз данных организована в виде словаря в том смысле, что база данных отображает ключи на их значения. Основное отличие в том, что база данных хранится на диске (или другом постоянном хранителе информации) и поэтому не исчезает после окончания программы. Также, поскольку база данных хранится на постоянном носителе, она способна вместить намного больший объем данных, чем словарь, размер которого ограничивается объемом оперативной памяти компьютера.

Как и в случае словаря, программы работы с базами данных разработаны так, чтобы добавление данных и доступ к ним происходили максимально быстро даже при их огромном объеме. Это достигается с помощью создания индексов, позволяющих компьютеру быстро перейти к конкретному элементу.

Существует множество различных систем для создания баз данных разных типов, включающее Oracle, MySQL, Microsoft SQL Server, PostgreSQL и SQLite. В этой книге мы рассмотрим SQLite, поскольку это очень распространенная база данных, поддержка которой встроена в Питон.

База SQLite специально сконструирована так, чтобы ее можно было включать в другие приложения, обеспечивая в их рамках поддержку баз данных. Например, браузер Firefox использует внутри себя SQLite, как и многие другие программы.

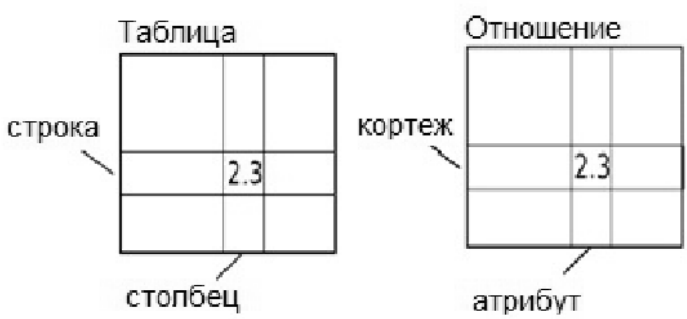
ссылка: http://sqlite.org/

SQLite хорошо подходит для решения разных задач, связанных с манипуляциями данными, например, при создании пауков Твиттера,

которых мы опишем в этой главе.

26.2. Понятия, относящиеся к базам данных

При первом взгляде на базу данных она выглядит как таблица с множеством листов. Первичная структура данных базы использует таблицы, которые состоят из строк и столбцов.

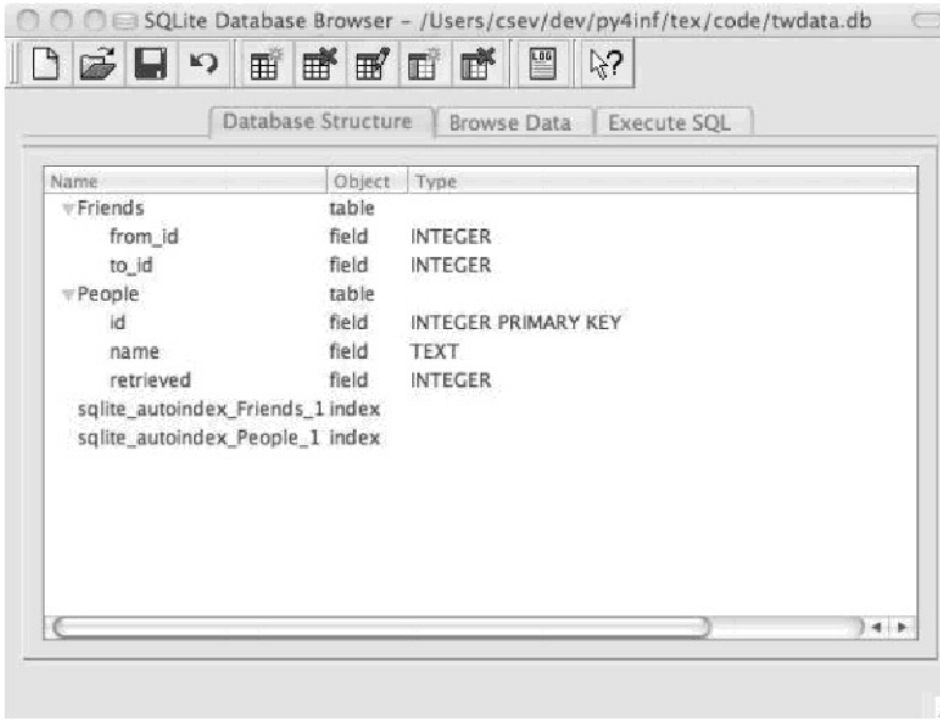


В техническом описании реляционных баз данных вместо не очень строгих понятий таблицы, строки и столбца используются более четко определенные формальные термины: отношение (relation), кортеж (tuple) и атрибут (attribute) соответственно. Мы все же будем пользоваться неформальными терминами в этой главе.

26.3. Браузер базы данных SQLite

Хотя в данной главе в основном рассматривается работа с базами данных SQLite из программ Питона, многие операции удобнее выполнять с помощью графической программы, которая называется "Браузер базы данных SQLite" (SQLite Database Browser) – это свободная программа, доступная по адресу ссылка: http://sourceforge.net/projects/sqlitebrowser/.

Используя браузер, вы легко можете создавать таблицы, добавлять и редактировать данные и выполнять простые SQL-запросы к базе данных:



Браузер базы данных в некотором смысле аналогичен текстовому редактору, работающему с текстовыми файлами. Когда нужно выполнить одну или несколько операций с текстовым файлом, можно просто открыть его в текстовом редакторе и внести нужные изменения. Но когда подобных изменений очень много, часто проще написать небольшую программу на Питоне. То же самое верно и при работе с базами данных: простые операции можно делать в браузере, но более сложные гораздо удобнее выполнять с помощью программ на Питоне.

26.4. Создание таблицы базы данных

Базы данных требуют более четкой структуры, чем списки или словари Питона[1]. При создании таблицы базы данных нужно заранее указать название каждого столбца и тип данных, которые мы собираемся хранить в столбце. Когда для программного обеспечения известны типы данных в столбцах, оно может выбрать наиболее эффективный способ их хранения и поиска.

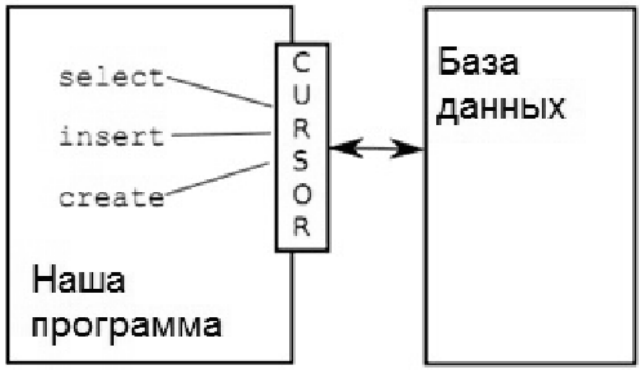
Можно посмотреть список различных типов данных, поддерживаемых SQLite, по адресу ссылка: http://www.sqlite.org/datatypes.html.

Определение структуры данных на начальном шаге может показаться неудобным, но зато мы получаем в результате быстрый доступ к данным даже в том случае, когда их объем очень большой.

Код для создания файла базы данных и ее таблицы с именем Tracks, которая содержит два столбца, выглядит следующим образом:

```
import sqlite3
conn = sqlite3.connect('music.db')
cur = conn.cursor()
cur.execute('DROP TABLE IF EXISTS Tracks ')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
conn.close()
```

Операция `connect` устанавливает "соединение" с базой данных, которая хранится в файле music.db в текущем каталоге. Если файл не существует, то он будет создан. Слово "соединение" используется потому, что достаточно часто база хранится на сетевом сервере, отличном от того компьютера, на котором работает приложение. Но в нашем простом случае база данных представляет собой локальный файл в том же каталоге, в котором мы выполняем программу Питона. Переменная `cur` играет роль файлового дескриптора, мы используем ее для операций с содержимым базы данных. Вызов метода `cursor()` концептуально близок к вызову `open()`, когда мы работаем с текстовыми файлами.



Как только мы получили дескриптор cur с помощью метода `cursor`, можно выполнять команды над содержимым базы данных при помощи метода `execute()`. Эти команды представляют собой специальный язык, который был стандартизован благодаря усилиям многих разработчиков различных систем баз данных – все системы теперь используют единый язык. Он называется "Язык структурированных запросов" (Structured Query Language) или сокращенно SQL.

ссылка: http://en.wikipedia.org/wiki/SQL

В нашем примере мы исполняем две SQL-команды в базе данных. По общему соглашению, принято записывать ключевые слова языка SQL прописными буквами, а части команды, которые мы добавляем к ключевым словам (например, имена таблицы и ее столбцов), – строчными буквами. Первая SQL-команда удаляет таблицу с именем Tracks из базы данных, если такая существует. Этот фрагмент кода позволяет многократно выполнять одну и ту же программу, которая каждый раз заново создает таблицу Tracks, избегая ошибок. Отметим, что команда DROP TABLE удаляет из базы данных таблицу со всем ее содержимым без возможности восстановления (отмена операции – "Undo" – не предусмотрена).

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

Вторая команда создает таблицу с именем Tracks, которая содержит два столбца: в стобец с именем title помещается текстовая информация, в столбец с именем plays – целые числа.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

Теперь, когда мы создали таблицу с именем Tracks, мы можем поместить данные в эту таблицу с помощью операции INSERT языка SQL. Как и в предыдущем случае, мы начинаем с установки соединения с базой данных и получения курсора (аналога файлового дескриптора). После этого, используя курсор, мы можем выполнять SQL-команды.

Команда SQL INSERT указывает, какую именно таблицу мы используем, затем задает новую строку таблицы, перечисляя поля, которые мы хотим в нее включить (title, plays), и после ключевого слова VALUES – значения, которые мы хотим поместить в новую строку таблицы. Мы можем задать значения с помощью вопросительных знаков (?, ?), чтобы указать, что реальные значения передаются в виде кортежа ('My Way', 15) в качестве второго параметра метода `execute()`:

```
import sqlite3
conn = sqlite3.connect('music.db')
cur = conn.cursor()
cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
( 'Thunderstruck', 20 ) )
cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
( 'My Way', 15 ) )
conn.commit()
print 'Tracks:'
cur.execute('SELECT title, plays FROM Tracks')
for row in cur :
print row
cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()
cur.close()
```

Сначала с помощью команды INSERT мы вставляем две строки в нашу таблицу, затем мы используем метод `commit()` для форсированной записи данных в файл.

Friends

title	plays
Thunderstruck	20
My Way	15

Затем мы используем команду SELECT, чтобы получить из таблицы две строки, которые только что были добавлены в нее. В команде SELECT мы указываем, какие столбцы нам нужны (title, plays), а также имя таблицы, из которой мы извлекаем информацию. После выполнения операции SELECT курсор (т.е. переменная `cur`) позволяет нам перебирать выбранные данные в цикле `for`. Для эффективности курсор в действительности не читает все данные из базы сразу при выполнении операции SELECT, вместо этого каждая очередная порция данных считывается по отдельности, когда мы перебираем выбранные строки в цикле `for`.

На выходе программы получаем:

```
Tracks:
(u'Thunderstruck', 20)
(u'My Way', 15)
```

В цикле `for` найдены две строки, каждая из которых представляет собой кортеж в смысле Питона, его первым элементом является заголовок (музыкального произведения), вторым – число его исполнений. Пусть вас не смущает префикс "u", с которого начинаются заголовки, – это просто указание, что строки представлены в кодировке Unicode, которая дает возможность использовать любые символы, а не только латинские буквы. В самом конце программы мы выполняем SQL-команду DELETE, удаляя только что созданные строки, что позволяет нам исполнять программу снова и снова.

Команда DELETE демонстрирует использование условия WHERE, задающего критерий выбора строк, к которым применяется команда. В

данном примере получилось так, что критерий применим ко всем строкам, в результате таблица становится пустой. Благодаря этому программу можно запускать много раз.

После выполнения команды DELETE мы также вызываем метод `commit()` для форсированного удаления данных из файла базы данных.

26.5. Обзор Языка структурированных запросов (SQL)

До сих пор мы использовали Язык структурированных запросов (Structured Query Language) в примерах программ на Питоне и изучили многие базовые SQL-команды. В этом разделе мы более детально рассмотрим язык SQL и дадим краткий обзор его синтаксиса.

Поскольку существует множество различных поставщиков баз данных, Язык структурированных запросов (SQL) был стандартизирован, чтобы мы могли единым образом взаимодействовать с различными системами баз данных многих поставщиков. Реляционная база данных состоит из таблиц, строк и столбцов. Типы данных в столбцах – это обычно текст, числа или даты. При создании таблицы мы указываем названия и типы данных в столбцах:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

Чтобы вставить строку в таблицу, мы используем SQL-команду INSERT:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

Команда INSERT указывает название таблицы, затем перечисляется список полей (названий столбцов), которые мы хотим задать в новой строке, потом после ключевого слова VALUES задается список значений соответствующих полей.

SQL-команда SELECT используется для извлечения строк и столбцов из базы данных.

Оператор SELECT позволяет указать, какие столбцы необходимо вывести, а условие WHERE задает критерий для выбора строк.

Необязательные ключевые слова ORDER BY позволяет задать способ сортировки полученных строк.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

Использование звездочки * указывает, что нужно возвратить все столбцы для каждой строки базы данных, удовлетворяющей условию WHERE.

Обратите внимание, что, в отличие от Питона, в SQL-условии WHERE мы используем одинарный, а не двойной знак равенства при проверке на равенство. В условии WHERE можно также указывать другие логические операции, используя знаки сравнения <, >, <=, >=, !=, а также ключевые слова AND, OR и круглые скобки для построения сложных логических выражений. Можно также отсортировать возвращенные строки по одному из полей:

```
SELECT title,plays FROM Tracks ORDER BY title
```

Чтобы удалить строки, нужно указать условие WHERE в SQL-операторе DELETE. Условие WHERE определяет, какие именно строки необходимо удалить:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

Можно обновить столбец или несколько столбцов внутри одной или более строки, используя оператор UPDATE языка SQL:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

В команде UPDATE сначала указывается таблица, затем после ключевого слова SET – список полей и их новых значений, и далее после ключевого слова WHERE следует необязательное условие, задающее выбор строк, которые должны быть обновлены. Один оператор UPDATE меняет сразу все строки, которые отвечают критерию выбора, указанному в WHERE, либо, если WHERE не используется, то обновляются вообще все строки в таблице.

Эти четыре основные команды (INSERT, SELECT, UPDATE и DELETE) позволяют выполнять четыре главные операции, необходимые для

создания данных и работы с ними.

26.6. Создание пауков Твиттера с использованием базы данных

В этом разделе мы создадим простую программу-паука, которая пройдет по всем учетным записям Твиттера и создаст по ним базу данных. Замечание: будьте осторожны, запуская эту программу! Не следует извлекать чересчур много данных или запускать программу на слишком долгое время, что может повлечь закрытие вашего аккаунта.

Одна из проблем, с которой мы сталкиваемся, когда создаем программу-паука – нужно иметь возможность в любой момент остановить ее и вновь запустить, это может повторяться многократно и при этом не должны теряться полученные ранее данные.

Не хотелось бы каждый раз вновь запускать процесс получения данных с самого начала, поэтому мы сохраняем данные сразу по их получении, таким образом, программа при перезапуске начинает работать с того места, где она была прервана.

Мы начинаем с какого-то пользователя Твиттера, извлекая список его друзей и их статусов, перебираем элементы этого списка в цикле и добавляем каждого из друзей в базу данных, чтобы в будущем извлечь и списки их друзей. После того, как мы обработали друзей одного человека, мы проверяем нашу базу данных и извлекаем из Твиттера друзей какого-нибудь человека, ранее занесенного в базу.

Это повторяется снова и снова, каждый раз мы находим в базе человека, которого еще "не посетили" в Твиттере, извлекаем список его друзей и добавляем в базу тех из них, кто ранее еще не был в нее занесен, чтобы в будущем посетить их тоже. По ходу дела мы отслеживаем также, сколько раз конкретный человек встретился в списках друзей, чтобы определить степень его "популярности".

Сохраняя в базе данных список известных учетных записей, а также для каждой записи информацию о том, был ли для нее извлечен список друзей или еще нет, плюс данные о том, насколько популярна эта учетная запись, мы получаем возможность остановить нашу программу

и вновь запустить ее столько раз, сколько нам захочется.

Эта программа довольно сложная. Она основана на упражнении, приведенном ранее в этой книге, в котором мы используем API Твиттера. Вот исходный код нашего Твиттер-паука:

```
import sqlite3
import urllib
import xml.etree.ElementTree as ET
TWITTER_URL = 'http://api.twitter.com/l/statuses/friends/ACCT.xml'
conn = sqlite3.connect('twdata.db')
cur = conn.cursor()
cur.execute('''
CREATE TABLE IF NOT EXISTS
Twitter (name TEXT, retrieved INTEGER, friends INTEGER)''')
while True:
acct = raw_input('Enter a Twitter account, or quit: ')
if ( acct == 'quit' ) : break
if ( len(acct) < 1 ) :
cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
try:
acct = cur.fetchone()[0]
except:
print 'No unretrieved Twitter accounts found'
continue
url = TWITTER_URL.replace('ACCT', acct)
print 'Retrieving', url
document = urllib.urlopen (url).read()
tree = ET.fromstring(document)
cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )
countnew = 0
countold = 0
for user in tree.findall('user'):
friend = user.find('screen_name').text
cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
(friend, ) )
try:
count = cur.fetchone()[0]
cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
```

```
  (count+1, friend) )
  countold = countold + 1
  except:
  cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
  VALUES ( ?, 0, 1 )''', ( friend, ) )
  countnew = countnew + 1
  print 'New accounts=',countnew,' revisited=',countold
  conn.commit()
  cur.close()
```

Наша база данных хранится в файле twdata.db, там содержится одна таблица с именем Twitter, содержащая три столбца: текстовый столбец "name" для имени аккаунта; целочисленный столбец "retrieved", содержащий единицу для тех аккаунтов, список друзей который уже был извлечен, либо ноль в противном случае; и целочисленный столбец "friends", содержащий количество записей, которые "подружились" с данным аккаунтом.

В основном цикле нашей программы запрашивается название Твиттер-аккаунта или слово "quit" для завершения программы. Если вводится название аккаунта Твиттера, мы извлекаем список его друзей и их статусы и добавляем каждого друга в базу данных, если он еще туда не внесен. Если он уже содержится в базе, то мы увеличиваем число его друзей на единицу.

Если пользователь просто нажал клавишу "Enter", то программа ищет в базе данных следующий аккаунт Твиттера, для которого список друзей еще не был получен, извлекает список его друзей, добавляет их в базу либо обновляет строку в базе, увеличивая счетчик друзей.

Как только мы получаем список друзей и их статусов, мы перебираем в цикле все элементы с тегом "user" полученного XML-документа и для каждого из них извлекаем текстовое значение подчиненного элемента "screen_name". Затем мы используем оператор SELECT для проверки, была ли запись с именем, содержащимся в "screen_name", ранее уже добавлена в базу, и для получения числа ее друзей (столбец "friends"), если запись уже была добавлена.

```
  countnew = 0
```

```
countold = 0
for user in tree.findall('user'):
friend = user.find('screen_name').text
cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
(friend, ) )
try:
count = cur.fetchone()[0]
cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
(count+1, friend) )
countold = countold + 1
except:
cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
VALUES ( ?, 0, 1 )''', ( friend, ) )
countnew = countnew + 1
print 'New accounts=',countnew,' revisited=',countold
conn.commit()
```

После выполнения команды SELECT мы должны извлечь выбранные из базы строки. Можно было бы сделать это, применяя цикл `for` к переменной `cur`, но, поскольку мы ограничили количество извлеченных строк единицей (LIMIT 1), можно использовать метод `fetchone()` ("выбрать один") для извлечения единственной строки, полученной в результате операции SELECT.

Поскольку метод `fetchone()` возвращает строку в виде кортежа (даже в том случае, когда строка содержит только одно поле), мы берем первое значение из кортежа, используя индексатор [0], и помещаем текущее значение счетчика друзей в переменную `count`.

Если выбор был успешным, то мы выполняем SQL-команду UPDATE с условием WHERE, чтобы увеличить на единицу значение в столбце "friends" той записи, которая соответствует аккаунту друга. Отметим, что в SQL-команде используются два подстановочных символа – вопросительные знаки, которые заменяются на реальные значения, передаваемые в виде двухэлементного кортежа в качестве второго параметра метода `execute()`.

Если исполнение кода внутри блока `try` приводит к неудаче, то это

происходит скорее всего потому, что в базе нет записей, подходящих под условие "WHERE name = ?" оператора SELECT. Поэтому в блоке `except`, обрабатывающем ошибочную ситуацию, мы используем SQL-команду INSERT, добавляя имя друга (полученное как `screen_name`) в таблицу с указанием, что список его друзей еще не извлечен (поле "retrieved" нулевое) и число друзей (поле "friends") равно нулю.

Запустив программу в первый раз и введя название учетной записи Твиттера, получим:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/l/statuses/friends/drchuck.xml
New accounts= 100 revisited= 0
Enter a Twitter account, or quit: quit
```

Поскольку мы запускаем программу в первый раз, база данных отсутствует, поэтому мы создаем ее в файле twdata.db и добавляем в нее таблицу с именем Twitter. Затем мы извлекаем нескольких друзей и помещаем всех их в базу данных, поскольку она изначально пуста.

Здесь мы хотели бы написать простую программу, распечатывающую текущее состояние базы, чтобы посмотреть содержимое нашего файла twdata.db:

```
import sqlite3
conn = sqlite3.connect('twdata.db')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur :
print row
count = count + 1
print count, 'rows.'
cur.close()
```

Эта программа открывает базу данных, выбирает все столбцы и все строки из таблицы Twitter и затем в цикле печатает каждую строку. Если мы выполним программу после первого запуска рассмотренного выше

паука Твиттера, то она напечатает следующее:

```
(u'opencontent', 0, 1)
(u'lhawthorn', 0, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
100 rows.
```

Для каждого имени аккаунта (полученного как XML-элемент screen_name) печатается одна строка, в которой указано, что мы еще не получили список друзей для данного имени (второй элемент 0) и что у имени есть 1 друг (третий элемент).

В данный момент содержание нашей базы отражает извлечение списка друзей для нашего первого аккаунта (drchuck). Мы можем снова запустить нашу программу, указав ей извлечь друзей первого "необработанного" аккаунта в базе простым нажатием клавиши "Enter" вместо ввода имени Твиттер-аккаунта:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/l/statuses/friends/opencontent.xml
New accounts= 98 revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/l/statuses/friends/lhawthorn.xml
New accounts= 97 revisited= 3
Enter a Twitter account, or quit: quit
```

Поскольку мы нажали "Enter" (т.е. не ввели название Твиттер-аккаунта), выполняется следующий фрагмент кода:

```
if ( len(acct) < 1 ) :
cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
try:
acct = cur.fetchone()[0]
except:
print 'No unretrieved twitter accounts found'
```

```
continue
```

Мы используем SQL-команду SELECT для получения имени первого (LIMIT 1) пользователя, у которого признак того, что мы извлекли его друзей (поле "retrieved"), все еще равен нулю. Также мы используем блок `try/except` и фрагмент `fetchone()[0]` внутри `try` для извлечения значения элемента `screen_name` из полученных данных; при ошибке печатается сообщение о том, что в базе уже нет необработанных записей. Если мы успешно получили имя еще необработанного аккаунта, мы извлекаем из Твиттера его данные следующим образом:

```
url = TWITTER_URL.replace('ACCT', acct)
print 'Retrieving', url
document = urllib.urlopen (url).read()
tree = ET.fromstring(document)
cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )
```

Получив успешно данные, мы выполняем SQL-команду UPDATE, чтобы заменить для обработанной записи значение поля "retrieved" на единицу – это означает, что мы уже извлекли список друзей данного аккаунта. Таким способом мы предотвращаем повторное извлечение из Твиттера уже обработанных данных, что позволяет каждый раз продвигаться вперед в Твиттере по сети друзей.

Если мы запустим программу и нажмем Enter дважды, чтобы извлечь друзей следующей необработанной записи, а затем распечатаем содержимое базы, то получим следующий вывод:

```
(u'opencontent', 1, 1)
(u'lhawthorn', 1, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
(u'cnxorg', 0, 2)
(u'knoop', 0, 1)
```

```
(u'kthanos', 0, 2)
(u'LectureTools', 0, 1)
...
295 rows.
```

Как мы видим, содержимое базы правильно отражает тот факт, что мы обработали аккаунты opencontent и lhawthorn. Отметим также, что аккаунты cnxorg и kthanos имеют двух друзей. На данный момент мы получили из сети друзей трех человек (drchuck, opencontent и lhawthorn), при этом наша таблица содержит 295 строчек (293 необработанных).

Каждый раз, когда мы запускаем программу, она находит следующий необработанный аккаунт (например, в нашем случае это steve_coppin), извлекает по сети его друзей, отмечает его как обработанный и для каждого друга аккаунта steve_coppin либо добавляет его в базу, либо увеличивает счетчик его друзей, если аккаунт друга уже содержится в базе.

Поскольку данные программы сохраняются в базе данных на диске, работа паука может быть приостановлена и возобновлена многократно без потери данных. Замечание. Прежде чем завершить эту тему, еще раз предупреждаем, что с программой-пауком Твиттера нужно быть осторожным. Не следует извлекать из сети слишком много данных или запускать программу на большое время – это может привести к потере доступа к Твиттеру.

26.7. Основы моделирования данных

Настоящая сила реляционных баз данных проявляется, когда мы создаем несколько таблиц и устанавливаем связи между ними. Принятие решений о том, каким именно образом разделить данные приложения между несколькими таблицами и как установить соотношения между двумя таблицами, называется моделированием данных (data modeling). Документ с дизайном вашего приложения, который показывает таблицы и их связи, называется моделью данных (data model).

Моделирование данных – непростое искусство, в этом разделе мы познакомимся лишь с самыми основами моделирования реляционных

данных. Более подробную информацию по этой теме можно найти по адресу ссылка: http://en.wikipedia.org/wiki/Relational_model.

Пусть в нашей программе-пауке Твиттера вместо простого подсчета друзей каждого человека мы хотим получить список всех входящих связей, чтобы можно было получить список всех тех, кто дружит с конкретным аккаунтом.

Поскольку каждый пользователь Твиттера может иметь множество аккаунтов, которые с ним дружат, недостаточно просто добавить единственный столбец в нашу таблицу Twitter. Поэтому мы создаем новую таблицу, в которой будут храниться пары друзей. Ниже указан простой способ создания подобной таблицы Pals (англ. "приятели"):

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

Каждый раз, когда мы встречаем человека, который является другом пользователя drchuck, мы добавляем в таблицу Pals строку вида:

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

После обработки 100 друзей аккаунта drchuck мы добавим в таблицу 100 записей, в которых "drchuck" будет первым параметром, что приведет нас к многократному повторению одной и той же строки в базе данных.

Такое повторение строковых данных нарушает хорошую практику нормализации баз данных (database normalization), которая требует, чтобы одна и та же строка не помещалась в базу дважды. Если нам нужно использовать данные более одного раза, то мы создаем целочисленный ключ к реальным данным и ссылаемся на них с помощью этого ключа.

Говоря практически, строка занимает намного больше пространства в памяти или на диске, чем целое число, и требует больше процессорного времени для сравнения и сортировки. Когда мы имеем всего лишь несколько сотен записей, объем памяти и время обработки несущественны. Но если в нашей базе данных миллион человек и, возможно, 100 миллионов ссылок на друзей, скорость сканирования данных становится очень важной.

Мы будем хранить наши Твиттер-аккаунты в таблице с именем People

вместо таблицы Twitter из предыдущих примеров. Таблица People имеет дополнительный столбец для хранения целочисленного ключа, соответствующего строке пользователя Твиттера. SQLite дает возможность автоматически добавлять ключевое значение для любой строки, добавляемой в базу данных, используя специальный тип данных в столбце (INTEGER PRIMARY KEY).

Таблица People с дополнительным столбцом, хранящим идентификаторы, создается следующим образом:

```
CREATE TABLE People
(id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

Отметим, что мы больше не поддерживаем счетчик числа друзей для каждой строки в таблице People. Когда мы задали тип столбца "id" как INTEGER PRIMARY KEY, мы указали, что SQLite сам должен позаботиться о содержимом этого столбца и автоматически назначить уникальный целочисленный ключ для каждой строки, которая добавляется в базу. Мы также использовали ключевое слово UNIQUE для того, чтобы запретить SQLite помещать в таблицу два разные строки с одним и тем же значением поля "name".

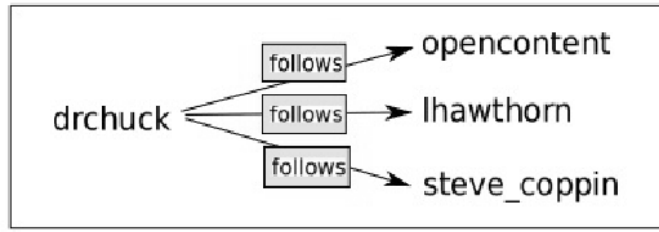
Теперь вместо того, чтобы создавать рассмотренную выше таблицу Pals, мы создадим таблицу Follows с двумя целочисленными столбцами "from_id" и "to_id" и тем ограничением, что комбинация двух чисел `from_id` и `to_id` должна быть уникальна в таблице (т.е. ее строки не могут повторяться).

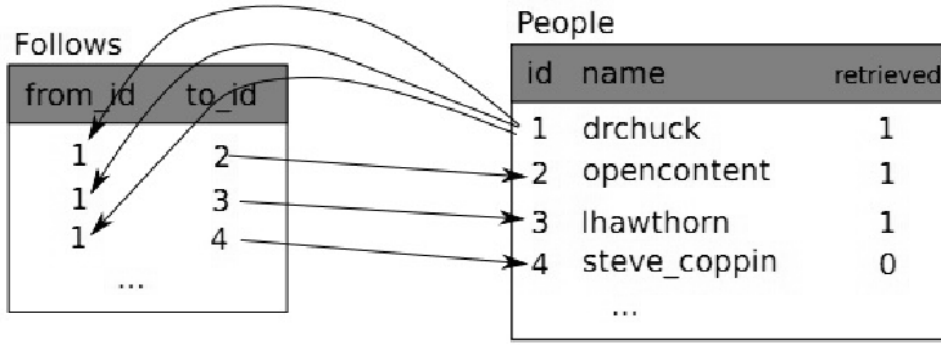
```
CREATE TABLE Follows
(from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

Добавляя условие UNIQUE при создании таблицы, мы устанавливаем ряд правил, действующих при добавлении записей в базу данных. Они делают программирование более удобным, во-первых, предотвращая возможные ошибки, и, во-вторых, упрощая код.

По существу, создавая таблицу Follows, мы моделируем "отношение", при котором один человек следует за некоторым другим, и представляем это

отношение парами чисел. Каждая пара а) указывает, что люди связаны между собой и (б) устанавливает направление этой связи.





26.8. Программирование с несколькими таблицами

Теперь мы перепишем нашу программу-паука Твиттера, используя две таблицы, первичные ключи и ссылки между ключами, как было описано выше. Вот код новой версии программы:

```
import sqlite3
import urllib
import xml.etree.ElementTree as ET
TWITTER_URL = 'http://api.twitter.com/l/statuses/friends/ACCT.xml'
conn = sqlite3.connect('twdata.db')
cur = conn.cursor()
cur.execute('''CREATE TABLE IF NOT EXISTS People
(id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
(from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
while True:
```

```
acct = raw_input('Enter a Twitter account, or quit: ')
if ( acct == 'quit' ) : break
if ( len(acct) < 1 ) :
cur.execute('''SELECT id,name FROM People
WHERE retrieved = 0 LIMIT 1''')
try:
(id, acct) = cur.fetchone()
except:
print 'No unretrieved Twitter accounts found'
continue
else:
cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
(acct, ) )
try:
id = cur.fetchone()[0]
except:
cur.execute('''INSERT OR IGNORE INTO People
(name, retrieved) VALUES ( ?, 0)''', ( acct, ) )
conn.commit()
if cur.rowcount != 1 :
print 'Error inserting account:',acct
continue
id = cur.lastrowid
url = TWITTER_URL.replace('ACCT', acct)
print 'Retrieving', url
document = urllib.urlopen (url).read()
tree = ET.fromstring(document)
cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ) )
countnew = 0
countold = 0
for user in tree.findall('user'):
friend = user.find('screen_name').text
cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
(friend, ) )
try:
friend_id = cur.fetchone()[0]
countold = countold + 1
except:
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
```

```
  VALUES ( ?, 0)''', ( friend, ) )
  conn.commit()
  if cur.rowcount != 1 :
  print 'Error inserting account:',friend
  continue
  friend_id = cur.lastrowid
  countnew = countnew + 1
  cur.execute('''INSERT OR IGNORE INTO Follows
  (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
  print 'New accounts=',countnew,' revisited=',countold
  conn.commit()
  cur.close()
```

Программа становится немного более сложной, но зато она иллюстрирует шаблоны кода, который используется при установлении связей между таблицами с помощью целочисленных ключей. Наиболее важные моменты следующие.

1. Создание таблиц с первичными ключами и ограничениями.
2. Когда мы имеем логический ключ, определяющий человека (в данном случае это имя аккаунта), нам нужно получить его id (т.е. соответствующий целочисленный ключ). В зависимости от того, занесен ли данный человек в таблицу People или еще нет, нам нужно либо (1) найти человека в таблице и извлечь значение id, либо (2) добавить человека в таблицу People и получить сгенерированное значение id для добавленной строки.
3. Добавление строки в таблицу Follows, устанавливающей соотношения между людьми.

Ниже мы рассмотрим каждый из этих пунктов.

26.8.1. Ограничения в таблицах баз данных

Поскольку мы уже разработали дизайн наших таблиц, мы можем сообщить исполняющей системе базы данных, что нужно установить ряд правил при работе с таблицами. Эти правила предотвращают возможные ошибки и добавление в базу некорректных данных. Вот код

для создания таблиц:

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
(id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
(from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

Используя ключевое слово UNIQUE, мы указываем, что значения в столбце "name" таблицы People должны быть уникальными (не могут повторяться). Точно так же уникальными должны быть и пары чисел в строках таблицы Follows. Это предотвращает такие ошибки, как добавление одного и того же соотношения дважды. Мы можем воспользоваться преимуществами этих ограничений в следующем коде:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
VALUES ( ?, 0)''', ( friend, ) )
```

Мы добавили условие OR IGNORE ("или игнорировать") в оператор INSERT, чтобы указать, что, если выполнение команды INSERT приведет к нарушению правила "поле name должно быть уникальным", исполняющая система должна проигнорировать эту команду.

Мы используем ограничения как страховочную сетку, чтобы случайно не сделать что-нибудь неправильно. Аналогично, следующий код предохраняет нас от двукратного добавления одного и того же соотношения в таблицу Follows:

```
cur.execute('''INSERT OR IGNORE INTO Follows
(from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

Снова мы указываем исполняющей системе базы данных, что она должна игнорировать наши попытки выполнения команды INSERT, если это приводит к нарушению условия уникальности для строк таблицы Follows.

26.8.2. Получение и добавление записи

Когда мы запрашиваем название аккаунта Твиттера, то, если аккаунт существует, нужно найти значение его id. Если такого аккаунта в таблице People еще нет, нужно его добавить и получить значение id для добавленной строки.

Это очень часто встречающийся фрагмент кода, например, он дважды использован в приведенной выше программе. Код демонстрирует, как находить id для аккаунта друга, когда мы извлекаем текстовое значение узла screen_name, подчиненного узлу user в XML-документе.

Поскольку со временем вероятность того, что аккаунт уже занесен в базу данных, возрастает, мы сначала проверяем, содержится ли соответствующая запись в таблице People, используя оператор SELECT. Если код внутри блока try выполняется нормально[2], мы получаем запись, используя метод `fetchone()`, затем извлекаем первый (и единственный) элемент возвращенного кортежа и записываем его в поле friend_id. Если операция SELECT заканчивается неудачно, то код `fetchone()[0]` приводит к отказу и управление передается в секцию `except`.

```
friend = user.find('screen_name').text
cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
(friend, ) )
try:
friend_id = cur.fetchone()[0]
countold = countold + 1
except:
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
VALUES ( ?, 0)''', ( friend, ) )
conn.commit()
if cur.rowcount != 1 :
print 'Error inserting account:',friend
continue
friend_id = cur.lastrowid
countnew = countnew + 1
```

Если мы попадаем в блок `except`, это означает, что строка не была найдена и нужно ее добавить. Мы используем команду INSERT OR

IGNORE, чтобы избежать ошибок, и затем вызываем метод `commit()` для форсированного обновления базы. После окончания записи можно проверить значение переменной `cur.rowcount`, чтобы посмотреть, сколько строк обновилось. Поскольку мы делали попытку добавить единственную строку, то, если число обновленных строк отлично от единицы, это свидетельствует об ошибке.

Если команда INSERT завершается успешно, мы можем использовать переменную `cur.lastrowid`, чтобы получить значение id, сгенерированное базой данных для созданной строки.

26.8.3. Хранение ссылок на друзей

Когда нам уже известны значения ключей пользователя Твиттера и его друга, указанного в XML, нетрудно добавить пару чисел в таблицу Follows с помощью следующего кода:

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?
(id, friend_id) )
```

Заметим, что мы поручили самой базе данных следить за тем, чтобы задающая отношение дружбы пара не была добавлена в таблицу дважды – для этого при создании таблицы мы задали ограничение на единственность, а при добавлении использовали вариант OR IGNORE ("или игнорировать") команды INSERT. Вот пример выполнения программы:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/l/statuses/friends/drchuck.xml
New accounts= 100 revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/l/statuses/friends/opencontent.xml
New accounts= 97 revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/l/statuses/friends/lhawthorn.xml
```

```
New accounts= 97 revisited= 3
Enter a Twitter account, or quit: quit
```

Мы начали с аккаунта drchuck и затем дали возможность программе автоматически найти следующие два аккаунта и добавить их в базу данных.

Ниже показаны несколько первых строк в таблицах People и Follows после завершения этого запуска программы:

```
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
295 rows.

Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
300 rows.
```

Можно видеть значения полей id, name, и visited в таблице People, а также пары чисел, задающие отношение дружбы, в таблице Follows. Из таблицы People видно, что мы посетили первых трех человек и что их данные уже получены из Твиттера. Данные в таблице Follows показывают, что пользователи 2-6 являются друзьями пользователя drchuck (его номер 1). Произошло это потому, что первыми были получены и помещены в базу друзья пользователя drchuck.

Если бы мы напечатали больше строк таблицы Follows, то увидели бы также и друзей пользователей с номерами 2 и 3.

26.9. Три вида ключей

Теперь, когда мы начали построение модели данных, помещая данные в несколько таблиц и связывая между собой строки этих таблиц с помощью ключей, нужно ознакомиться с терминологией, относящейся к ключам.

В общем случае имеется три вида ключей, используемых в моделях баз данных.

- Логический ключ (logical key) берется из "реальной жизни" и может использоваться при поиске строки. В нашем примере он содержится в поле "name". Это экранное имя (screen name) пользователя Твиттера, и мы действительно несколько раз ищем строку пользователя по этому имени в нашей программе. Чаще всего следует использовать ограничение UNIQUE (уникальный) для логического ключа. Поскольку логический ключ используется для поиска нужной строки, производимого извне, вряд ли имеет смысл хранить в таблице несколько строк с одним и тем же значением ключа.
- Первичный ключ (primary key) – это целое число, автоматически назначенное базой данных для данной строки. Вне программы оно обычно не имеет никакого смысла и используется только для того, чтобы связывать между собой строки из разных таблиц. Если мы хотим найти строку в таблице, то поиск по первичному ключу – обычно самый быстрый из всех возможных. Поскольку первичные ключи являются целыми числами, они требуют минимальной памяти для хранения и могут сравниваться и сортироваться очень быстро. В нашей модели первичный ключ содержится в поле "id".
- Внешний ключ (foreign key) – это обычно число, указывающее на первичный ключ строки из другой таблицы. Примером внешнего ключа в нашей модели данных является содержимое поля "from_id". Мы придерживаемся следующего соглашения об именах ключей: поле, содержащее первичный ключ, всегда имеет имя "id"; поле, содержащее внешний ключ, образуется путем добавления к "id" спереди некоторого префикса и символа подчеркивания, оно имеет вид "prefix_id".

26.10. Использование команды JOIN для получения данных

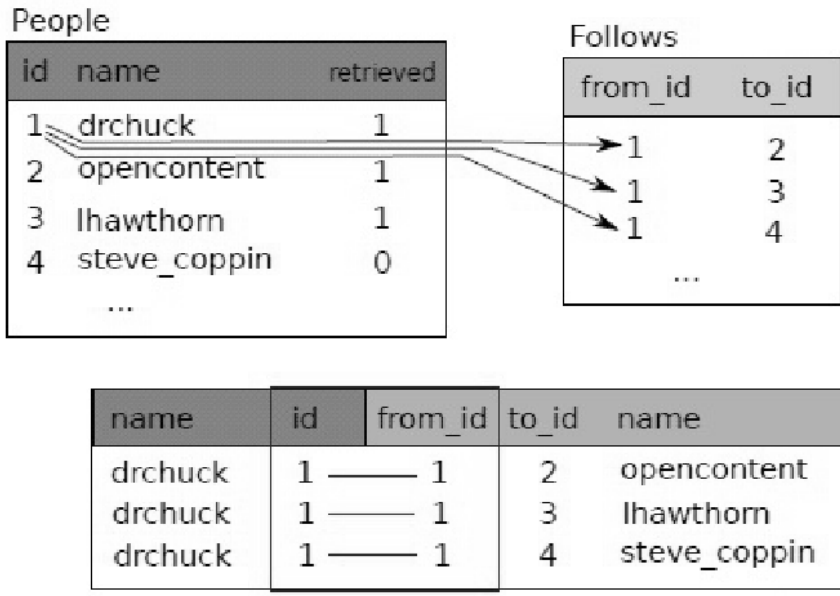
Теперь, когда мы, следуя правилу о нормализации базы данных, разделили данные на две таблицы, связанные с помощью первичных и внешних ключей, нам нужен способ выбора данных в команде SELECT, который мог бы собрать вместе данные из разных таблиц.

SQL использует условие JOIN для соединения таблиц. В нем указываются поля, которые служат для соединения строк из разных таблиц.

Приведем пример команды SELECT с условием JOIN:

```
SELECT * FROM Follows JOIN People
ON Follows.to_id = People.id WHERE Follows.from_id = 2
```

Условие JOIN указывает, что мы выбираем поля сразу из двух таблиц: Follows и People. Условие ON задает, как именно соединяются две таблицы. Берем строки из таблицы Follows и добавляем в их концы строки из таблицы People, у которых значение поля "id" совпадает со значением поля "from_id" строки из Follows.



Результатом команды JOIN является создание сверхдлинных "мета-строк", которые содержит как поля из таблицы People, так и соответствующие поля из таблицы Follows. Когда есть больше одного совпадения значений полей "id" таблицы People и "from_id" таблицы Follows, команда JOIN создает несколько мета-строк, соответствующих каждой совпадающей паре ключей, дублируя данные при необходимости.

Следующий пример распечатывает данные, которые мы имеем в базе после того, как приведенный выше вариант программы Твиттер-паука, основанный на двух таблицах, был запущен несколько раз.

```
import sqlite3
conn = sqlite3.connect('twdata.db')
cur = conn.cursor()
cur.execute('SELECT * FROM People')
count = 0
print 'People:'
for row in cur :
if count < 5: print row
count = count + 1
print count, 'rows.'
```

```
cur.execute('SELECT * FROM Follows')
count = 0
print 'Follows:'
for row in cur :
if count < 5: print row
count = count + 1
print count, 'rows.'
cur.execute('''SELECT * FROM Follows JOIN People
ON Follows.to_id = People.id WHERE Follows.from_id = 2''')
count = 0
print 'Connections for id=2:'
for row in cur :
if count < 5: print row
count = count + 1
print count, 'rows.'
cur.close()
```

Программа сначала распечатывает содержимое таблиц People и Follows и затем печатает часть данных из этих таблиц, соединенных вместе.

Вот вывод этой программы:

```
python twjoin.py
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
295 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
300 rows.
Connections for id=2:
```

```
(2, 1, 1, u'drchuck', 1)
(2, 28, 28, u'cnxorg', 0)
(2, 30, 30, u'kthanos', 0)
(2, 102, 102, u'SomethingGirl', 0)
(2, 103, 103, u'ja_Pac', 0)
100 rows.
```

Вначале идут данные таблиц People и Follows; последние строки вывода представляют собой результат выполнения команды SELECT с условием JOIN. В ней мы находим аккаунты, которые являются друзьями аккаунта "opencontent" (т.е. People.id=2).

В каждой "мета-строке", возвращенной последней командой SELECT, первые два поля получены из таблицы Follows, за ними следуют поля с третьего по пятое из таблицы People. Можно также заметить, что в каждой объединенной "мета-строке" второе поле (Follows.to_id) соответствует третьему полю (People.id).

26.11. Резюме

В этой главе мы познакомились с основами использования баз данных в программах на Питоне. Это сложнее, чем использовать для хранения данных словари Питона или обычные файлы, поэтому работать с базами данных следует только тогда, когда это действительно необходимо. Базы данных могут оказаться весьма полезными в следующих ситуациях:

1. когда приложению требуется сделать несколько изменений внутри большого набора данных,
2. когда объем данных настолько велик, что он не помещается в словарь, и при этом поиск информации часто повторяется,
3. когда работает некоторый долговременный процесс, который может быть остановлен и возобновлен, при этом данные должны сохраняться между перезапусками.

Во многих приложениях достаточно простой базы данных с единственной таблицей, но все же большинство реальных задач

требуют использования нескольких таблиц и связей между строками разных таблиц.

Когда возникает необходимость вводить связи между таблицами, важно хорошо продумать их дизайн и следовать правилу нормализации баз данных, чтобы обеспечить максимально эффективное использование их возможностей.

Поскольку чаще всего необходимость в базах данных возникает из-за огромного объема данных, с которыми приходится иметь дело, важно построить эффективную модель данных, чтобы программа работала максимально быстро.

26.12. Отладка

Когда вы разрабатываете программу на Питоне, использующую базу данных SQLite, распространенным способом отладки является просмотр содержимого базы с помощью браузера базы данных SQLite ("SQLite Database Browser"). После запуска вашей программы браузер дает возможность быстро проверить, правильно ли работает ваша программа.

Нужно учитывать, что система SQLite предотвращает одновременное изменение одних и тех же данных разными программами. Например, если вы открыли базу данных в браузере, сделали какое-то изменение и всё ещё не нажали клавишу "save" (сохранить), браузер "блокирует" (lock) доступ к файлу базы для любых других программ.

В частности, ваша программа на Питоне не сможет работать с файлом базы, когда он заблокирован. Решение состоит в том, чтобы либо закрыть браузер, либо, используя его меню "File", закрыть базу данных перед тем, как начать опять работать с ней из программы Питона, – это позволит избежать отказов программы из-за блокировки файла базы.

26.13. Глоссарий

Атрибут (attribute): одно из значений внутри кортежа. Чаще используются термины "столбец" или "поле".

Ограничение (constraint): указание базе данных, что к полю или к строке таблицы применяется некоторое правило. Чаще всего используется ограничение, требующее, чтобы не было дублирования значений в конкретном поле (т.е. все значения должны быть уникальными).

Курсор (cursor): позволяет выполнять SQL-команды над содержимым базы данных и извлекать данные из базы. В применении к базе данных курсор является аналогом файлового дескриптора в случае обычного файла или сокета в случае сети.

Браузер базы данных (database browser): программа, дающая возможность прямого подсоединения к базе данных, просмотра и изменения ее содержимого без необходимости написания программного кода.

Внешний ключ (foreign key): целочисленный ключ, который ссылается на первичный ключ некоторой строки в другой таблице. Внешние ключи устанавливают связи между строками разных таблиц.

Индекс (index): дополнительные данные, которые программное обеспечение баз данных поддерживает при добавлении строк в таблицу; они используются для ускорения поиска.

Логический ключ (logical key): ключ, используемый для поиска конкретной строки из "внешнего мира". Например, в таблице, содержащей учетные записи пользователей, адрес электронной почты человека является хорошим кандидатом на роль логического ключа.

Нормализация (normalization): создание модели данных таким образом, чтобы исключить дублирование данных. Мы храним каждый элемент данных только в одном месте, используя во всех других местах ссылки на него с помощью внешнего ключа.

Первичный ключ (primary key): целочисленный ключ, ассоциированный с каждой строкой таблицы, который используется для ссылки на данную строку из других таблиц. Часто база данных конфигурируется таким образом, чтобы автоматически генерировать первичные ключи при добавлении строк.

Отношение (relation): область внутри базы данных, содержащая кортежи

и атрибуты. Чаще используется термин "таблица".

Кортеж (tuple): одна запись в таблице базы данных, представляющая собой набор атрибутов. Чаще используется термин "строка".

26.14. Упражнения

Упражнение 26.1.

Получите по сети файл ссылка: http://www.py4inf.com/code/wikidata.db и используйте браузер базы данных SQLite, чтобы узнать, сколько таблиц содержится в базе; определите также для каждой таблицы список ее полей и их типов. Тип одного из полей не был рассмотрен в этой главе. Используйте online-документацию SQLite, чтобы описать, для чего нужен подобный тип данных.

[1] SQLite на самом деле допускает некоторую гибкость при задании типов данных в столбцах, но в этой главе мы ограничимся более строгим подходом, который применим и к другим системам баз данных, например, к MySQL.

[2] Как правило, если предложение начинается со слов "если всё выполняется нормально", то обычно код требует включения его внутрь блока `try/except`.

Автоматизация типичных задач на вашем компьютере

Мы уже считывали данные из файлов, сетевых сервисов и баз данных. Питон также может пройти по всем каталогам и папкам вашего компьютера и прочитать содержащиеся в них файлы. В этой главе мы напишем программы, сканирующие компьютер и выполняющие некоторые операции над каждым файлом.

Файлы размещены в каталогах (которые также называют "директориями" или "папками"). Простой скрипт на Питоне способен выполнить работу, которую необходимо сделать над сотнями и тысячами файлов, содержащихся в дереве каталогов всего компьютера.

Чтобы пройти все каталоги и файлы в дереве директорий, мы используем метод `os.walk` и цикл `for`. Аналогичным образом обычная команда `open` дает возможность прочитать в цикле содержимое файла, сокет позволяет в цикле читать данные через сетевое соединение, а библиотека `urllib` дает возможность открыть веб-документ и в цикле просматривать его содержимое.

28.1. Имена файлов и пути

Каждая работающая программа имеет "текущий каталог" (current directory), который является каталогом по умолчанию для большинства операций. Например, когда мы открываем файл для чтения, Питон ищет его в текущем каталоге. Модуль `os` (сокращение от "operating system" – операционная система) обеспечивает нас функциями для работы с файлами и каталогами; метод `os.getcwd` возвращает название текущего каталога:

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/Users/csev
```

Аббревиатура `cwd` является сокращением от "current working directory". В приведенном примере результатом является /Users/csev, это домашняя

директория пользователя с именем csev.

Подобная строка, идентифицирующая файл, называется путем (path). Относительный путь начинается в текущем каталоге; абсолютный стартует с корневой директории файловой системы.

Пути, с которыми мы имели дело до сих пор, были попросту именами файлов, т.е. они были относительными (рассматривались файлы в текущей директории). Для нахождения абсолютного пути к файлу можно воспользоваться методом `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/Users/csev/memo.txt'
```

Метод `os.path.exists` проверяет, существует ли файл или каталог:

```
>>> os.path.exists('memo.txt')
True
```

Если путь существует, то метод `os.path.isdir` определяет, задает ли он каталог (директорию):

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Аналогично, метод `os.path.isfile` проверяет, задает ли он файл.

Метод `os.listdir` возвращает список всех файлов и каталогов в заданном каталоге:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

28.2. Пример: очистка каталога с именем "photo"

Некоторое время назад я написал программу, похожую на Flickr, которая получает фотографии с моего мобильного телефона и сохраняет их на моем сервере. Я написал ее еще до того, как появилась Flickr, и продолжаю использовать ее и после появления Flickr, поскольку я хочу хранить оригинальные копии моих фотографий бесконечно долго.

Я также посылаю однострочное текстовое описание фотографии в MMS-сообщении или в строке subject (тема) сообщения электронной почты. Это сообщение сохраняется в текстовом файле, расположенном в том же каталоге, что и файл с изображением. Я использовал структуру каталогов, основанную на месяце, годе, дне и времени создания фотографии. Ниже приведен пример названий для одной фотографии и её описания:

```
./2006/03/24-03-06_2018002.jpg
./2006/03/24-03-06_2018002.txt
```

После семи лет я накопил огромное количество фотографий и их описаний. С течением времени я менял мои мобильные телефоны, и иногда мой код, извлекающий описания фотографий из сообщений, приводил к ошибкам и добавлял массу бессмысленной информации на сервер вместо подписей к фотографиям.

Мне хотелось бы просмотреть все такие файлы и определить, какие из них в действительности содержат подписи и какие – лишь мусор, чтобы удалить испорченные файлы. Первым делом можно написать простую программу, подсчитывающую число текстовых файлов в подкаталогах текущего каталога:

```
import os
count = 0
for (dirname, dirs, files) in os.walk('.'):
for filename in files:
if filename.endswith('.txt') :
count = count + 1
print 'Files:', count
python txtcount.py
Files: 1917
```

Ключевым элементом в этой программе является вызов метода `os.walk` библиотеки Питона. Когда мы вызываем метод `os.walk` и указываем ему стартовый каталог, он "обходит" все каталоги и подкаталоги внутри начального рекурсивно. Строка "." обозначает текущий каталог, который нужно пройти "в глубину". В процессе обхода в цикле for для каждого каталога мы получаем три значения в форме кортежа: его первым элементом является имя очередного каталога, вторым элементом – список его подкаталогов, третьим – список его файлов.

Нам не нужно явно исследовать содержимое каждого подкаталога, поскольку можно положиться на метод `os.walk`, который рано или поздно посетит каждый подкаталог. Но нам необходимо проверить все файлы, поэтому мы написали простой цикл `for` для проверки каждого файла в очередном каталоге. Мы проверяем, заканчивается ли имя файла на ".txt" и, если да, увеличиваем счетчик числа файлов, имена которых оканчиваются суффиксом ".txt".

Как только мы подсчитали общее число файлов с суффиксом ".txt", следующим шагом будет попытка автоматически определить с помощью Питона, какие файлы плохие и какие хорошие. Напишем простую программу, печатающую для каждого файла путь к нему и его размер.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
for filename in files:
if filename.endswith('.txt') :
thefile = os.path.join(dirname,filename)
print os.path.getsize(thefile), thefile
```

Теперь вместо простого подсчета файлов мы создаем строку, представляющую путь к файлу, путем соединения имени директории с именем файла внутри нее при помощи метода `os.path.join`. Важно использовать именно `os.path.join` вместо простой конкатенации строк, поскольку в Windows в обозначении пути к файлу используется символ "обратная косая черта" (backslash '\'), а в операционных системах Linux и Apple – прямая косая черта '/'. Метод `os.path.join` знает об

этих различиях и учитывает, в какой системе работает программа, выбирая правильный разделитель; поэтому программа Питона работает правильно и под Windows, и в системах типа Unix.

После того, как мы получили полное имя файла, включающее путь к нему, мы используем метод `os.path.getsize`, чтобы получить и напечатать размер файла. Вот что выдает наша программа:

```
python txtsize.py
...
18 ./2006/03/24-03-06_2303002.txt
22 ./2006/03/25-03-06_1340001.txt
22 ./2006/03/25-03-06_2034001.txt
...
2565 ./2005/09/28-09-05_1043004.txt
2565 ./2005/09/28-09-05_1141002.txt
...
2578 ./2006/03/27-03-06_1618001.txt
2578 ./2006/03/28-03-06_2109001.txt
2578 ./2006/03/29-03-06_1355001.txt
...
```

Рассматривая вывод, обратим внимание на то, что некоторые файлы совсем короткие, а некоторые, наоборот, очень большие, причем они имеют один и тот же размер (либо 2578, либо 2565). Посмотрев содержимое одного из подобных файлов, можно увидеть, что в них не содержится ничего, кроме одинакового HTML-текста, присланного моей системе с моего мобильного телефона:

```
<html>
<head>
<title>T-Mobile</title>
...
```

Просматривая дальше файл, мы не находим никакой содержательной информации в нем, поэтому такие файлы, возможно, следует удалить. Но перед удалением файлов мы напишем программу, которая находит файлы, имеющие внутри более одной строки, и печатает их

содержимое. Также мы не будем нагружать себя рассмотрением файлов размером в точности 2578 или 2565 символов, поскольку мы уже знаем, что эти файлы заведомо не содержат полезной информации.

Итак, напишем следующую программу:

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
for filename in files:
if filename.endswith('.txt') :
thefile = os.path.join(dirname,filename)
size = os.path.getsize(thefile)
if size == 2578 or size == 2565:
continue
fhand = open(thefile,'r')
 lines = list()
for line in fhand:
lines.append(line)
fhand.close()
if len(lines) > 1:
print len(lines), thefile
print lines[:4]
```

Мы используем оператор `continue` для пропуска файлов с одним из двух "плохих" размеров, остальные файлы открываем и считываем содержащиеся в них строки в список Питона; если строк больше одной, мы печатаем количество строк в файле и первые 3 строки.

Всё это выглядит как отфильтровывание файлов с двумя плохими размерами, а также допущение, что файлы, содержащие только одну строку, корректны; после этого выдача нашей программы становится достаточно содержательной:

```
python txtcheck.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
```

```
3 ./2007/09/15-09-07_074202_03.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/19-09-07_124857_01.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/20-09-07_115617_01.txt
...
```

Остался еще один тип файлов, доставляющих беспокойство: это файлы, содержащие по 3 строки, из которых первые две пустые, а третья строка представляет собой сообщение "Sent from my iPhone" ("Отправлено с моего телефона"), неизвестно как просочившееся внутрь моих данных. Поэтому мы сделаем еще одно изменение в нашей программе, чтобы учесть и такие файлы.

```
lines = list()
for line in fhand:
lines.append(line)
if len(lines) == 3 and lines[2].startswith('Sent from my iPhone') :
continue
if len(lines) > 1:
print len(lines), thefile
print lines[:4]
```

Мы просто проверяем файлы из трех строк, и, если третья строка начинается с указанного текста, пропускаем файл.

Теперь, запустив программу, мы видим всего 4 оставшихся многострочных файла, и все 4 выглядят вполне разумно:

```
python txtcheck2.py

3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
2 ./2006/03/17-03-06_1806001.txt
['On the road again...\r\n', '\r\n']
2 ./2006/03/24-03-06_1740001.txt
```

```
['On the road again...\r\n', '\r\n']
```

Посмотрев еще раз на процесс разработки этой программы, мы видим, как последовательно улучшается множество приемлемых файлов: отыскав очередной шаблон "плохих" файлов, мы пропускаем их с помощью оператора `continue`, что позволяет на следующем шаге найти еще один плохой шаблон.

Теперь мы готовы удалить все плохие файлы, поэтому изменим логику программы на противоположную: вместо печати оставшихся "хороших" файлов мы будет печатать "плохие" файлы, которые мы планируем удалить.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
for filename in files:
if filename.endswith('.txt') :
thefile = os.path.join(dirname,filename)
size = os.path.getsize(thefile)
if size == 2578 or size == 2565:
print 'T-Mobile:',thefile
continue
fhand = open(thefile,'r')
lines = list()
for line in fhand:
lines.append(line)
fhand.close()
if len(lines) == 3 and lines[2].startswith('Sent from my iPhone') :
print 'iPhone:', thefile
continue
```

Мы получили список файлов, являющихся кандидатами на удаление, причем для каждого файла указана причина, по которой его следует удалить. Вот вывод программы:

```
python txtcheck3.py
...
```

```
T-Mobile: ./2006/05/31-05-06_1540001.txt
T-Mobile: ./2006/05/31-05-06_1648001.txt
iPhone: ./2007/09/15-09-07_074202_03.txt
iPhone: ./2007/09/15-09-07_144641_01.txt
iPhone: ./2007/09/19-09-07_124857_01.txt
...
```

Можно еще раз выборочно проверить эти файлы, чтобы убедиться, что мы не сделали ошибки в нашей программе и она не найдет файлы, которые не хотелось бы удалять. Если мы удовлетворены этой проверкой, внесем следующие изменения в программу:

```
if size == 2578 or size == 2565:
print 'T-Mobile:',thefile
os.remove(thefile)
continue
...
if len(lines) == 3 and lines[2].startswith('Sent from my iPhone') :
print 'iPhone:', thefile
os.remove(thefile)
continue
```

В этом варианте программы мы печатаем названия плохих файлов и затем удаляем их, используя метод `os.remove`.

```
python txtdelete.py
T-Mobile: ./2005/01/02-01-05_1356001.txt
T-Mobile: ./2005/01/02-01-05_1858001.txt
...
```

Ради интереса запустите программу во второй раз – она не выдаст ничего, поскольку все плохие файлы уже уничтожены. Если запустить рассмотренную ранее программу txtcount.py, подсчитывающую текстовые файлы, мы увидим, что было удалено 899 плохих файлов:

```
python txtcount.py
Files: 1018
```

В этом разделе мы выполняли следующие шаги: сначала использовали Питон для просмотра всех директорий и файлов в них, пытаясь найти шаблоны нежелательных файлов. Затем, находя очередной шаблон, мы улучшали результаты поиска, что в конце концов помогло точно определить, какие именно файлы мы хотим удалить. Наконец, на последнем шаге мы с помощью Питона удалили все ненужные файлы.

Задача определения требуемого множества файлов может быть совсем простой и зависеть, например, только от имен файлов, – но, возможно, нам придется считывать содержимое каждого файла и искать какие-либо текстовые фрагменты внутри него. Иногда приходится читать все файлы и вносить изменения в некоторые из них. В любом случае всякая подобная задача легко решается, когда мы понимаем, как работает метод `os.walk` и другие методы из библиотеки `os`.

28.3. Аргументы командной строки

В предыдущих главах мы рассмотрели ряд программ, которые запрашивали у пользователя имя файла, используя функцию `raw_input`, и затем читали и обрабатывали данные из файла:

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
...
```

Можно несколько упростить подобные программы, получая имя файла из командной строки, которая используется при запуске программы на Питоне. До сих пор мы просто запускали программу и отвечали на ее запросы:

```
python words.py
Enter file: mbox-short.txt
...
```

В командной строке можно указать дополнительные подстроки после имени файла с программой Питона, их обычно называют аргументами командной строки. Вот простая программа, демонстрирующая чтение аргументов из командной строки:

```
import sys
print 'Count:', len(sys.argv)
print 'Type:', type(sys.argv)
for arg in sys.argv:
print 'Argument:', arg
```

Содержимое переменной `sys.argv` является списком строк, в котором первая строка – это имя Питон-программы, а следующие строки представляют собой аргументы командной строки, указанные в команде после имени файла с программой. Ниже приведен вывод нашей программы для конкретной командной строки:

```
python argtest.py hello there
Count: 3
Type: <type 'list'>
Argument: argtest.py
Argument: hello
Argument: there
```

Здесь 3 аргумента командной строки передаются нашей программе в виде трехэлементного списка. Первым элементом является имя программы (argtest.py), двумя другими (hello и there) – слова, указанные в команде после имени файла.

Можно переписать нашу программу, чтобы она читала файл, получая его имя из командной строки:

```
import sys
name = sys.argv[1]
handle = open(name, 'r')
text = handle.read()
print name, 'is', len(text), 'bytes'
```

В качестве имени файла берется второй аргумент командной строки (пропускается имя программы, соответствующее индексу [0]). Мы открываем файл, читаем его содержимое и печатаем его длину в байтах:

```
python argfile.py mbox-short.txt
mbox-short.txt is 94626 bytes
```

Использование аргументов командной строки облегчает повторное использование Питон-программ, особенно когда нужно вводить только одну или две строки.

28.4. Программные каналы (pipes)

Большинство операционных систем предоставляет интерфейс командной строки, известный под названием оболочка (shell). Оболочка обычно предоставляет команды для перемещения по файловой системе и запуска приложений. Например, в Unix'е можно перемещаться по директориям с помощью команды "cd", просматривать содержимое директории с помощью "ls" и запускать веб-браузер, например, с помощью команды "firefox".

Любая программа, которую можно запустить из командной оболочки, может быть запущена также и из программы Питона с использованием канала.

Программный канал (pipe) – это объект, представляющий работающий процесс.

Например, команда Unix'а[1] "ls -l" показывает содержимое текущего каталога (в подробном формате). Можно запустить эту команду, используя метод `os.popen`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

Аргументом является строка, содержащую команду оболочки. Возвращаемое значение является указателем на файл, который можно

использовать точно так же, как и при открытии обычного файла с помощью функции `open`. Можно читать вывод процесса "ls" последовательно по одной строке с помощью метода `readline` или получить сразу весь вывод с помощью метода `read`:

```
>>> res = fp.read()
```

По окончании работы следует закрыть канал так же, как и файл:

```
>>> stat = fp.close()
>>> print stat
None
```

Возвращаемое методом `close` значение содержит статус завершения процесса `ls`; "None" означает нормальное завершение (т.е. отсутствие ошибок).

28.5. Глоссарий

Абсолютный путь (absolute path): строка, описывающая, где хранится файл или каталог (директория), начинающаяся с корня дерева каталогов. Абсолютный путь можно использовать для доступа к файлу или каталогу независимо от текущего каталога.

Контрольная сумма (checksum): см. также "хеширование". Термин "контрольная сумма" был порожден необходимостью проверки данных, посланных по сети или записанных на внешний носитель и затем прочитанных обратно. Когда данные записываются или пересылаются, передающая система вычисляет контрольную сумму и пересылает ее вместе с данными. Когда данные считываются или принимаются по сети, принимающая система перевычисляет контрольную сумму полученных данных и сравнивает ее с принятой контрольной суммой. Если контрольные суммы не совпадают, то это означает, что данные были искажены при передаче.

Аргументы командной строки (command line arguments): параметры, указанные в командной строке Питона после имени файла с программой.

Текущий каталог/директория (current working directory): текущий каталог, в котором "вы находитесь". Можно изменить текущий каталог, используя команду "cd", которая есть в большинстве операционных систем в командном интерфейсе. Когда вы открываете файл в Питоне, используя только его имя и не указывая путь, файл должен быть в текущем каталоге, в котором вы запускаете программу.

Хеширование (hashing): чтение потенциально очень большого объема данных и вычисление контрольной суммы для этих данных — так называемой хеш-функции. Лучшие хеш-функции создают минимальное число "коллизий", когда два различных потока данных дают при вычислении хеш-функции один и тот же результат. MD5, SHA1 и SHA256 являются названиями наиболее распространенных хеш-функций.

Программный канал (pipe): устанавливает связь между работающими программами. Используя канал, можно написать программу, которая посылает данные другой программе или принимает данные от нее. Программный канал аналогичен сокету, за исключением того, что каналы могут использоваться лишь для связи между программами, работающими на одном и том же компьютере (не через сеть).

Относительный путь (relative path): строка, описывающая, где хранится файл или каталог (директория) относительно текущего каталога.

Командная оболочка (shell): интерфейс командной строки к операционной системе, называемый также "терминалом" в некоторых системах. В нем пользователь вводит команду и ее параметры и затем нажимает клавишу "Enter" для выполнения команды.

Обход (walk): термин, используемый для описания процесса посещений узлов дерева каталогов, подкаталогов, под-подкаталогов, пока мы не посетим все каталоги. Мы называем этот процесс "обходом дерева каталогов/директорий".

28.6. Упражнения

Упражнение 28.1.

В большом собрании MP3-файлов могут быть копии одних и тех же песен, сохраненные в разных директориях или в файлах с разными именами. Цель этого упражнения – найти все повторяющиеся файлы.

1. Напишите программу, которая обходит все каталоги и подкаталоги, находит все файлы с указанным суффиксом (например, .mp3) и перечисляет пары файлов с одинаковым размером. Совет: используйте словарь, в котором ключом является размер файла, полученный с помощью метода `os.path.getsize`, а значением является путь к файлу (включая его имя). При получении очередного файла проверяйте, имеется ли уже в словаре файл с таким же размером. Если да, то надо напечатать размер файла и названия обоих файлов (один из словаря, второй — текущий просматриваемый файл).
2. Измените предыдущую программу так, чтобы она сравнивала не только размеры, но и содержимое файлов, используя алгоритм вычисления контрольной суммы или хеш-функции. Например, алгоритм MD5 (Message-Digest algorithm 5) читает "сообщение" произвольной длины и вычисляет 128-битовую "контрольную сумму". Вероятность того, что у двух разных файлов будет одинаковая контрольная сумма, ничтожно мала. Описание MD5 можно прочитать по адресу ссылка: wikipedia.org/wiki/Md5. Следующий фрагмент кода открывает файл, читает его содержимое и вычисляет контрольную сумму.

```
import hashlib
...
fhand = open(thefile,'r')
data = fhand.read()
fhand.close()
checksum = hashlib.md5(data).hexdigest()
```

Вы должны создать словарь, в котором контрольная сумма используется как ключ, а имя файла — как значение ключа. Если вычисленная контрольная сумма файла уже содержится в словаре в виде ключа, значит, найдены два файла с одинаковым содержимым; поэтому мы печатаем путь к файлу из словаря и к текущему рассматриваемому файлу. Вот что выдает программа,

запущенная для каталога с файлами изображений:

```
./2004/11/15-11-04_0923001.jpg ./2004/11/15-11-04_1016001.jpg
./2005/06/28-06-05_1500001.jpg ./2005/06/28-06-05_1502001.jpg
./2006/08/11-08-06_205948_01.jpg ./2006/08/12-08-06_155318_02.jp
./2006/09/28-09-06_225657_01.jpg ./2006/09-50-years/28-09-06_225
./2006/09/29-09-06_002312_01.jpg ./2006/09-50-years/29-09-06_002
```

Очевидно, я иногда отправляю одни и те же фотографии более одного раза или копирую фотографии без удаления файла-оригинала.

[1] При использовании каналов для вызова команд операционной системы, таких, как "ls", важно знать, какую именно операционную систему вы используете, и вызывать только команды, поддерживаемые операционной системой.

Содержание