Ръководство по Питон

Издание 1.5.2

Гуидо ван Росум

22 март 2000

Corporation for National Research Initiatives
1895 Preston White Drive, Reston, VA 20191, USA
E-mail: guido@python.org

Резюме

Питон[1] е лесен за използване, мощен език за програмиране. Той разполага със структури от данни от високо ниво и прост, но ефективен подход към обектно-ориентираното програмиране. Елегантният синтаксис на Питон, динамичните му типове, заедно с интерпретируемостта му го правят идеален език за скриптиране и бърза разработка на приложения (RAD) в много области под повечето платформи.

Интерпретаторът на Питон и обширната му стандартна библиотека могат да се разпространяват безплатно и са достъпни на страницата на Питон `http://www.python.org` във вид на изходен или двоичен код за повечето основни платформи. Същата страница съдържа и дистрибуции на, и показалци към много безплатни модули на Питон от трети страни, програми, инструменти и допълнителна документация.

Интерпретаторът на Питон се разширява лесно с нови функции и типове данни реализирани на C или C++ (или други езици, които могат да се извикват от C). Питон е също така подходящ като разширителен език за гъвкави приложения.

Това ръководство запознава читателя с основните понятия и възможности на езика и системата Питон. То помага да се почувствате удобно пред клавиатурата и интерпретатора на Питон, но всички примери са самостоятелни и съдържателни, така че ръководството може да бъде четено и без интерпретаторът да Ви е подръка.

За описание на стандартните обекти и модули, вижте документа *Python Library Reference*. *Python Reference Manual* дава по-формална дефиниция на езика. За да пишете разширения на C или C++, прочетете наръчниците *Extending and Embedding* и *Python/C API*. Има и няколко книги, които подробно разглеждат Питон.

Това ръководство не се опитва да включи и обхване всяка отделна възможност или дори всяка често използвана възможност. Вместо това, то запознава с много от най-забележителните възможности на Питон, и ще Ви даде добра представа за аромата и стила на езика. След като го прочетете, Вие ще сте в състояние да четете и пишете модули и програми на Питон. Ще бъдете готови да научите повече за различните модули от библиотеката на Питон, описани в *Python Library Reference*.

[1] на английски се пише *Python* и се произнася *пайтън*. *(бел. прев.)*

Предговор към българското издание

Аз не съм добър преводач. Но предложеният превод е най-доброто, което засега мога да постигна. Затуй искам да поднеса пред читателя едно извинение и една молба.

Нека читателят да ми прости неминуемите за преводаческата ми култура грешки. Молбата ми е, ако някъде срещнете правописна грешка, или тромав, неподходящ израз, или такъв, който просто Ви дразни, не се сърдете. Но непременно ми пишете. Ще съм много благодарен на такива корекции и ги насърчавам с всички средства. Приемам всякакви критики – както общи, така и конкретни. Станете редактор на това ръководство, за да подробрим качеството му! В специален списък ще включвам редакторите, които са ми подсказали удачен превод на отделни пасажи, фрази или дори и на най-невзрачната дума.

Искам специално да благодаря на Елена Стойкова и Юнуз Юнуз за оказаната помощ при началното коригиране на превода.

Тъй като този превод се подобрява и обновява, последната му версия винаги е публично достъпна на адрес: http://python-bg.sourceforge.net.

Благодаря.

Калоян Доганов
е-поща: kaloian@users.sourceforge.net
12 февруари 2001

Списък на редакторите

(по азбучен ред)

- Елена Стойкова
- Любен Каравелов
- Никола Колев
- Светослав Николов
- Юнуз Юнуз

СЪДЪРЖАНИЕ

1 Да изострим апетита Ви **1**
1.1 Оттук накъде? . . . 2

2 Използване на интерпретатора на Питон **3**
2.1 Извикване на интерпретатора . . . 3
2.2 Интерпретаторът и неговото обкръжение . . . 4

3 Неформално въведение в Питон **7**
3.1 Използване на Питон като калкулатор . . . 7
3.2 Първи стъпки в програмирането . . . 13

4 Повече средства за управление на изпълнението **15**
4.1 `if`-конструкции . . . 15
4.2 `for`-конструкции . . . 15
4.3 Функцията `range()` . . . 16
4.4 Оператори `break` и `continue`, и клаузи `else` в циклите . . . 16
4.5 Операторът `pass` . . . 17
4.6 Дефиниране на функции . . . 17
4.7 Повече за дефинирането на функции . . . 19

5 Структури от данни **23**
5.1 Повече за списъците . . . 23
5.2 Операторът `del` . . . 25
5.3 Комплекти и редици . . . 26
5.4 Речници . . . 26
5.5 Повече за условията . . . 27
5.6 Сравняване на редици и други типове . . . 28

6 Модули **29**
6.1 Повече за модулите . . . 30
6.2 Стандартни модули . . . 31
6.3 Функцията `dir()` . . . 32
6.4 Пакети . . . 32

7 Вход и изход **37**
7.1 По-гиздаво форматиране на изхода . . . 37
7.2 Четене и писане на файлове . . . 39

8 Грешки и изключения **43**
8.1 Синтактични грешки . . . 43
8.2 Изключения . . . 43
8.3 Обработка на изключения . . . 44
8.4 Предизвикване на изключения . . . 45
8.5 Дефинирани от потребителя изключения . . . 46

8.6 Дефиниране на почистващи действия 46

9 Класове **47**
9.1 Няколко думи за терминологията 47
9.2 Обсеги и пространства на имената в Питон 48
9.3 Пръв поглед върху класовете 49
9.4 Нахвърляни бележки 51
9.5 Наследяване 52
9.6 Частни променливи 54
9.7 Туй-онуй 54

10 Сега какво? **57**

A Интерактивно редактиране на входа и заместване с история **59**
A.1 Редактиране на реда 59
A.2 Заместване с история 59
A.3 Значение на клавишните комбинации 59
A.4 Коментар 60

ГЛАВА
ПЪРВА

Да изострим апетита Ви

Ако някога сте писали голям скрипт за обвивката (shell script)[1], сигурно Ви е познато това чувство: ще е хубаво да добавите още едно свойство, но скриптът е вече толкова бавен, толкова голям и толкова сложен; а може би свойството повлича след себе си необходимостта от извикване на системна или друга функция, която е достъпна само от C. . . Обикновено наличният проблем не е достатъчно сериозен за да оправдае пренаписването на скрипта на C; може би проблемът изисква символни низове с променлива дължина или други типове данни (като сортирани списъци от файлови имена), които са лесни от обвивката, но изискват много работа за реализиране на C, или пък Вие не сте достатъчно запознати със C.

Друга ситуация: може би трябва да работите с няколко C библиотеки и обичайния цикъл на писане/компилиране/тестване/прекомпилиране на C е много бавен. Вие трябва да разработите софтуера по-бързо. Възможно е също да сте написали програма, която може да използва разширителен език, и не искате да проектирате език, да пишете и отстранявате грешките в интерпретатора за него, та чак после да го вградите в приложението си.

В случаи като тези, вероятно Питон е език точно за Вас. Питон е лесен за използване, но е истински език за програмиране, и предлага повече структура и поддръжка за големи програми, отколкото обвивката. От друга страна, той осигурява много повече проверки за грешки, отколкото C, и бивайки *език от много високо ниво*, той притежава вградени типове данни от високо ниво, като гъвкави масиви (arrays) и речници (dictionaries) които биха Ви коствали дни за да напишете ефикасно на C. Благодарение на по-общите си типове данни, Питон има много по-голяма област на приложение отколкото *Awk* или дори *Perl*, като в същото време много неща в Питон са поне толкова лесни, колкото и в тези езици.

Питон Ви позволява да разделите програмата си на модули, които могат да бъдат използвани отново в други програми на Питон. Той идва с голям набор от стандартни модули, които можете да използвате за основа на програми си, или като примери, от които можете да започнете да се учите да програмирате на Питон. Също така има и вградени модули, които обезпечават такива неща като файлов вход/изход (I/O), системни функции, сокети (sockets), и дори програмни интерфейси към GUI библиотеки като Tk.

Питон е интерпретируем език, който може да Ви спести значително време за разработка, защото не са необходими компилиране и свързване (linking). Интерпретаторът може да се използва интерактивно, което го прави лесен за експериментиране с възможностите на езика, за писане на програми "за боклука", или за тестване на функциите по време на разработка отдолу-нагоре. Той също е и удобен настолен калкулатор.

Питон позволява писането на много компактни и четими програми. Програмите, създадени на Питон обикновено са много по-кратки от еквивалентните, писани на C или C++, поради ред причини:

- типовете данни от високо ниво Ви позволяват да изразите сложни действия в един-единствен оператор;
- групирането на изразите се извършва чрез отстъп, вместо чрез начални/крайни скоби;
- не са необходими декларации на променливи или аргументи.

Питон е *разширяем*: ако можете да програмирате на C, тогава е лесно да добавите нова вградена функция или модул към интерпрететора, както и да извършите критични операции с максимална скорост, или да свържете програми на Питон към библиотеки, които могат да са достъпни само в двоичен вид (например някоя специфична за определен производител графична библиотека). Веднъж след като сте истински запалени, можете да свържете интерпретатора на Питон към приложение, писано на C, и да го използвате като разширителен или команден език за това приложение.

[1]За някои ключови думи в скоби са дадени английските термини, за да може читателят да се ориентира в англоезичната документация и литература за Питон. *(бел. прев.)*

Между другото, езикът е кръстен на шоуто на BBC “Monty Python’s Flying Circus” и няма нищо общо с гадните влечуги. Позоваването на пародиите на Monty Python в документацията е не само позволено, но и насърчително![2]

1.1. Оттук накъде?

Сега, след като Питон ви заинтригува, може би ще пожелате да го разгледате по-отблизо. Тъй като най-добрият начин да се научи един език е като се използва, тук вие сте поканени да го направите.

В следващата глава се обяснява механизмът за използване на интерпретатора. Наистина, това е по-скоро скучна информация, но тя е съществена за изпробването на примерите, показани по-късно.

Останалата част от ръководството чрез примери запознава с различни възможности на езика и системата Питон, като започва с прости изрази, оператори и типове данни, през функции и модули, и накрая засяга концепции за напреднали като изключения (exceptions) и дефинирани от потребителя класове (user-defined classes).

[2] В примерите на оригиналното ръководство се използват имена от Monty Python, но в българския превод те са са заменени с български. *(бел. прев.)*

ГЛАВА
ВТОРА

Използване на интерпретатора на Питон

2.1. Извикване на интерпретатора

На машините, които разполагат с интерпретатор на Питон, той е обикновено е инсталиран като '/usr/local/bin/python'; добавянето на '/usr/local/bin' в пътя за търсене на обвивката Ви позволява да го стартирате с въвеждането на командата

```
python
```

в обвивката. Тъй като изборът на директория за разполагане на интерпретатора е инсталационна опция, възможни са и други места; питайте местния Питон-гуру или системния администратор. (Популярна алтернатива е местоположението '/usr/local/python'.)

Въвеждането на знака EOF (Control-D в UNIX, Control-Z в DOS или Windows) след първоначалния промпт предизвиква изход от интерпретатора с нулев код за изход (exit status). Ако това не работи, можете да излезете от интерпретатора като въведете следните команди: '`import sys; sys.exit()`'.

Възможностите на интерпретатора за редактиране на текущия ред обикновено не са много сложни. В UNIX, който и да е инсталирал интерпретатора, може би е разрешил поддръжката за библиотеката GNU readline, която добавя разширени възможности за редактиране и прелистване назад в буфер от вече въведените команди. Може би най-бързата проверка за това дали се поддържа редактиране на командния ред е да натиснете Control-P на първия промпт на Питон, който получите. Ако чуете звуков сигнал, значи разполагате с редактиране на командния ред; вижте Приложение A за въведение в клавишните комбинации. Ако нищо не се случи, или пък се изпише `^P`, значи редактирането на командния ред не е достъпно; ще можете да използвате само клавиша за връщане назад за да изтривате знаците от текущия ред.

Интерпретаторът действа подобно на UNIX обвивка: когато е извикан със стандартен вход свързан към tty-устройство, той чете и изпълнява команди интерактивно; когато е извикан с файлово име като аргумент или с файл на стандартния вход, той чете и изпълнява *скрипта* от този файл.

Трети начин за стартиране на интерпретатора е '`python -c команда [арг] ...`', което изпълнява оператор(ите) в `команда`, аналогично на опцията `-c` на обвивката. Тъй като операторите на Питон често съдържат интервали или други знаци, които обвивката третира по особен начин, най-добре е да заградите цялата `команда` в двойни кавички.

Забележете, че има разлика между '`python file`' и '`python <file`'. Във втория случай, входните заявки от програмата, като извиквания на `input()` и `raw_input()`, се задоволяват от *файл*. Тъй като този файл вече е бил докрай прочетен от граматичния анализатор (parser) преди програмата да е започнала изпълнението си, то тя незабавно ще срещне EOF. В първия случай (който обикновено е този, който желаете), те се задоволяват от който и да е файл или което и да е устройство, свързани към стандартния вход на интерпретатора на Питон.

Когато се използва скрипт файл, понякога е удобна възможността да се изпълни скрипта и след това да се влезе в интерактивен режим. Това може да стане като се подаде `-i` преди скрипта. (Няма да се получи, ако скрипта се чете от стандартния вход, поради същите причини, обяснени в горния параграф.)

2.1.1. Предаване на аргументи

Когато са известни на интерпретатора, името на скрипта и аргументите към него се предават на скрипта в променливата `sys.argv`, която е списък от символни низове (string list). Този списък съ-

държа поне един елемент; когато не са дадени скрипт и аргументи, `sys.argv[0]` е празен символен низ. Когато името на скрипта е дадено като `'-'` (сиреч стандартен вход), `sys.argv[0]` е установен на `'-'`. Когато е използван форматът `-c команда`, тогава `sys.argv[0]` е установен на `'-c'`. Опциите, открити след `-c команда` не се третират като опции, предназначени за интерпретатора на Питон, а вместо това се разполагат в `sys.argv`, за да бъдат обработени от командата.

2.1.2. Интерактивен режим

Когато командите се четат от tty, това ще рече, че интерпретаторът се намира в *интерактивен режим*. В този режим той подканя за следващата команда чрез *първичен промпт*, който обикновено е три знака по-голямо (‘`>>> `’); за продължение на реда той подканя чрез *вторичен промпт*, обикновено три точки (‘`... `’).

Преди да изведе своя първичен промпт, интерпретаторът извежда поздравително съобщение, започвайки със собствения си номер на версия и бележка за авторско право.

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Допълнителните редове са необходими при въвеждането на многоредова конструкция. Например, погледнете този оператор `if`:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Внимавай да не паднеш!"
...
Внимавай да не паднеш!
```

2.2. Интерпретаторът и неговото обкръжение

2.2.1. Обработка на грешките

Когато се появи грешка, интерпретаторът извежда съобщение за грешка и трасиране на стека. Ако се намира в интерактивен режим, след това се връща към първичния промпт; а когато входът идва от файл, той излиза с ненулев код за изход след трасирането на стека. (В този смисъл, изключенията обработени с `except` клауза в `try` конструкция не са грешки.) Някои грешки са безусловно фатални и предизвикват изход от интерпретатора с ненулев код; това се отнася за вътрешните противоречия и някои случаи на изчепрване на паметта. Всички съобщения за грешка се пишат в стандартния поток (stream) за грешки; нормалният изход от изпълнените команди се пише в стандартния изход.

Въвеждането на знак за прекъсване (обикновено Control-C или DEL) в първичния или вторичния промпт отменя входа и връща към първичния промпт.[1] Отпечатването на прекъсване, когато се изпълнява команда, води до изключение `KeyboardInterrupt`, което може да бъде обработено от `try` оператор.

2.2.2. Изпълними скриптове на Питон

На BSD-подобните UNIX системи, скриптовете на Питон могат да бъдат направени директно изпълними подобно на скриптовете на обвивката, чрез разполагането на реда

```
#! /usr/bin/env python
```

(приемаме, че самият интерпретатор е в потребителския $PATH) в началото на скрипта и установяването на изпълним режим за файла. Тези ‘`#!`’ трябва да бъдат първите два знака от файла. Забележете, че знакът за диез, ‘`#`’, в Питон се използва за начало на коментар.

[1] Проблем с пакета GNU Readline може да попречи на това.

2.2.3. Интерактивен начален файл

Често, когато използвате Питон интерактивно, е удобно да разполагате със стандартни команди, които да се изпълняват всеки път, когато се стартира интепретатора. Можете да направите това като установите в променлива от обкръжението, наречена $PYTHONSTARTUP, името на файла, съдържащ Вашите начални команди. Това е подобно на свойството '.profile' на UNIX обвивките.

Този файл се прочита само в интерактивните сесии. Когато Питон чете командите от скрипт, или '/dev/tty' е даден като явен източник на команди, този файл не се прочита. Във всички останали случаи, интерпретаторът се държи като интерактивна сесия. Файлът се изпълнява в същото пространство на имена, където се изпълняват и интерактивните команди, така че обектите, които той дефинира или импортира, могат без явно определяне да се използват в интерактивната сесия. В този файл също така можете да смените промптовете `sys.ps1` и `sys.ps2`.

Ако искате да прочетете допълнителен начален файл от текущата директория, тогава можете да програмирате това в глобалния начален файл, напр. '`execfile('.pythonrc.py')`'. Ако искате да използвате началния файл в скрипт, трябва изрично да укажете това в скрипта:

```
import os
if os.environ.get('PYTHONSTARTUP') \
   and os.path.isfile(os.environ['PYTHONSTARTUP']):
    execfile(os.environ['PYTHONSTARTUP'])
```

ГЛАВА
ТРЕТА

Неформално въведение в Питон

В следващите примери, входът и изходът се разлчават по присъствието или отсъствието на промптовете (‘>>> ’ и ‘... ’): за да проиграете примера, когато промтът се появи трябва да напишете всичко, което се намира след него; редовете, които не започват с промпт, са отпечатани от интерпретатора.

Забележете, че наличието на самостоятелен вторичен промпт в даден пример означава, че трябва да въведете празен ред; той се използва за отбелязване на края на многоредова команда.

Много от примерите в този наръчник, дори и тези, които се въвеждат в интерактивния промпт, включват коментари. Коментарите в Питон започват със знак за диез, ‘#’, и продължават до края на физическия ред. Коментар може да се появи в началото на реда или след празно пространство или код, но не и в рамките на символен низ. Знакът за диез в символен низ е просто знак за диез.

Няколко примера:

```
# това е първият коментар
SPAM = 1                 # а това е вторият коментар
                         # ... а ето и трети!
STRING = "# Това не е коментар."
```

3.1. Използване на Питон като калкулатор

Нека изпробваме няколко прости команди на Питон. Стартирайте интерпретатора и изчакайте появяването на първичния промпт. (Това не би трябвало да отнеме много време.)

3.1.1. Числа

Интерпретаторът действа като прост калкулатор: можете да въвеждате изрази в него и той ще изписва стойността им. Синтаксисът на изразите е ясен: операторите +, -, * и / работят точно по същия начин, както и в повечето други езици (напр. Паскал или C); могат да се използват скоби за групиране. Например:

```
>>> 2+2
4
>>> # Това е коментар
... 2+2
4
>>> 2+2  # и коментар на същия ред заедно код
4
>>> (50-5*6)/4
5
>>> # Целочисленото деление връща цялата част:
... 7/3
2
>>> 7/-3
-3
```

Както в C, знакът за равенство (‘=’) се използва за присвояване на стойност на променлива. Стойността на присвояването не се изписва:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Една стойност може да бъде присвоена едновременно на няколко променливи:

```
>>> x = y = z = 0  # Нулираме x, y и z
>>> x
0
>>> y
0
>>> z
0
```

Съществува пълна поддръжка на плаваща запетая; операторите със смесен тип на операндите превръщат целочисления си операнд в тип с плаваща запетая:

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
```

Поддържат се също и комплексни числа; имагинерните числа се записват със суфикс '`j`' или '`J`'. Комплексните числа със ненулев реален компонент се записват като '(*реал*+*имаг*`j`)', или могат да бъдат създадени с функцията '`complex(`*реал*`, `*имаг*`)`'.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Комплексните числа са представени винаги като две числа с плаваща запетая, реална и имагинерна част. За да извлечете тези две части от комплексното число *z*, използвайте *z*`.real` и *z*`.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Преобразуващите функции към плаваща запетая и целочислен тип (`float()`, `int()` и `long()`) не работят с комплексни числа — няма правилен начин да се преобразува комплексно число в реално. Използвайте `abs(`*z*`)` за да вземете величината му (като плаваща запетая) или `z.real`, за да вземете реалната му част.

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

В интерактивен режим, последният изписан израз е присвоен на променливата _. Това означава, че когато използвате Питон като настолен калкулатор, ще малко по-лесно да продължите изчисленията, например:

```
>>> tax = 17.5 / 100
>>> price = 3.50
>>> price * tax
0.6125
>>> price + _
4.1125
>>> round(_, 2)
4.11
```

Потребителят трябва да разглежда тази променлива като "само за четене". Не ѝ присвоявайте стойност явно — тогава ще създадете независима локална променлива със същото име, скриваща вградената променлива с магическо поведение.

3.1.2. Символни низове

Освен с числа, Питон може да борави и със символни низове, които могат да бъдат изразени по няколко начина. Те могат да са затворени в единични или двойни кавички:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Символните низове могат да се простират на повече редове по няколко начина. Новите редове могат да се избегнат с обратни наклонени черти, т.е.:

```
hello = "Това е доста дълъг символен низ, съдържащ\n\
няколко реда текст така както бихте го направили в C.\n\
    Забележете, че празното пространство в началото на реда\
 има значение.\n"
print hello
```

Което ще отпечата следното:

```
Това е доста дълъг символен низ, съдържащ
няколко реда текст така както бихте го направили в C.
    Забележете, че празното пространство в началото на реда има значение.
```

Или пък, символните низове могат да бъдат затворени в тройни кавички: """ или '''. Когато използвате тройните кавички, няма нужда да избягвате края на редовете, но те ще бъдат включени в символния низ.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

предизвиква следния изход:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Интерпретаторът извежда резултата от операциите със символни низове по същия начин, по който те са въведени — в кавички, а кавичките и другите забавни знаци са предшествани от обратна наклонена черта, за да се покаже точната им стойност. Символният низ е затворен в двойни кавички, ако съдържа единична кавичка и не съдържа никакви двойни кавички. В противен случай символният низ е затворен в единични кавички. (Операторът `print`, описан по-долу, може да бъде използван за отпечатване на символни низове без кавички и обратно наклонени черти, предшестващи специалните знаци.)[1]

Символните низове могат да бъдат сливани (слепвани заедно) чрез оператора `+`, и повтаряни с `*`:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Два символни низа един до друг се събират автоматично: първия ред по-горе може да бъде написан като '`word = 'Help' 'A'`'; това работи само при два символни низа, но не и при произволни изрази със символни низове.

```
>>> 'str' 'ing'                   #  <-  Това е наред
'string'
>>> string.strip('str') + 'ing'   #  <-  Това е наред
'string'
>>> string.strip('str') 'ing'     #  <-  Това е невалидно
  File "<stdin>", line 1
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

Символните низове могат да бъдат индексирани; както в C, първият знак в един символен низ има индекс 0. Не съществува отделен тип данни за самостоятелен знак (char); знакът просто е символен низ с дължина едно. Подобно на езика Icon, подстринговете могат да бъдат определяни чрез *конвенцията за изрязване* — два индекса разделени с двоеточие.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

За разлика от C, символните низове в Питон не могат да бъдат променяни. Опитът за присвояване на индексирана позиция в символен низ предизвиква грешка:

[1] За съжаление, интерпретаторът на Питон включва буквите от кирилицата тъкмо към групата на "забавните" знаци. Така кирилицата се изписва нечетимо с кодове, предшествани от обратно наклонени черти. Този неприятен ефект не се отнася за действието на оператора `print`. Например, въвеждането на `'чорба'` в първичния промпт ще изведе `\347\256\340\241\240`, докато `print 'чорба'` ще изведе `чорба`. *(бел. прев.)*

```
>>> word[0] = 'x'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:-1] = 'Splat'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Обаче лесно и ефикасно можете да създадете нов символен низ, съдържащ желаното съчетание:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[-1:]
'SplatA'
```

При изрязването индексите имат полезни стойностти по подразбиране; изпуснат първи индекс се приема за нула, а изпуснат втори индекс приема стойност, равна на размера на символния низ, който се изрязва.

```
>>> word[:2]    # Първите два знака
'He'
>>> word[2:]    # Всичко освен първите два знака
'lpA'
```

Ето и едно полезно свойство на изрязващите операции: `s[:i] + s[i:]` е равно на `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Изродясалите индекси за изрязване се обработват с милост: индекс, който е твъде голям, се заменя с размера на символния низ, както и ако горната граница на идекса е по-малка от долната, се връща празен символен низ.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Индексите могат да имат отрицателни стойности, за да започнат броенето от дясно на ляво. Например:

```
>>> word[-1]     # Последният знак
'A'
>>> word[-2]     # Предпоследният знак
'p'
>>> word[-2:]    # Последните два знака
'pA'
>>> word[:-2]    # Всичко освен последните два знака
'Hel'
```

Забележете обаче, че -0 наистина е същото като 0, така че не се брои от дясно на ляво!

```
>>> word[-0]     # (тъй като -0 е равно на 0)
'H'
```

Негативните индекси за изрязване, излизащи извън рамките на символния низ, се окастрят. Но не опитвайте това с индексите за единичен елемент (т.е. когато не изрязвате).

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # грешка
Traceback (innermost last):
  File "<stdin>", line 1
IndexError: string index out of range
```

Най-добрият начин да запомните как работи изрязването е като мислите за индексите като за сочещи *между* знаците, като левия край на първия знак е номериран с 0. Тогава десният край на последния знак от символен низ, състоящ се от n знака има индекс n, например:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Първият ред от числа дава позицията на индексите 0...5 в символния низ; вторият ред дава съответните им отрицателни индекси. Изрезката от i до j съдържа всички знаци между краищата, отбелязани с i и j, респективно.

За неотрицателните индекси дължината на изрезката е разликата между индексите, ако и двата са в рамките на символния низ. Сиреч, дължината на `word[1:3]` е 2.

Вградената функция `len()` връща дължината на даден символен низ:

```
>>> s = 'непротивоконституционствуателствувайте'
>>> len(s)
39
```

3.1.3. Списъци

Питон познава няколко *съставни* (compound) типа данни, използвани за групиране на стойности. Най-гъвкавият от тях е *списъкът* (list), който може да бъде изписан като поредица от разделени с запетая стойности (елементи) между квадратни скоби. Не е нужно елементите от списъка да бъдат от един и същ тип.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Подобно на индексите на символните низове, индексите на списъците започват от 0, и списъците също могат да бъдат изрязвани, сливани и прочие:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

За разлика от символните низове, които са *непроменливи*, в списък е възмжно да се смени единичен елемент.

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Възможно е и присвояването на изрезки, като това дори може да промени размера на списъка:

```
>>> # Заменя някои елементи:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Премахва някои:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Вмъква някои:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a     # Вмъква (копие от) себе си в началото
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

Вградената функция `len()` важи и за списъците:

```
>>> len(a)
8
```

Възможно е списъците да се вмъкват (да се създават списъци, съдържащи други списъци), например:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Виж секция 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Забележете, че в посления пример `p[1]` и `q` действително отпращат към един и същ обект! По-късно ще се върнем отново на *обектната семантика*.

3.2. Първи стъпки в програмирането

Разбира се, можем да използваме Питон за по-сложни задачи от събиране на две и две. Например можем да изведем началната поредица от числата на *Фибоначи* ето така:

```
>>> # Числа на Фибоначи:
... # сумата на два елемента определя следващия
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Този пример въвежда няколко нови свойства.

– Първият ред съдържа *многократно присвояване*: променливите `a` и `b` едновременно получават новите стойности 0 и 1. В последния ред това е използвано отново, показвайки че изразите от дясната страна се изчисляват преди каквото и да било присвояване. Изразите от дясната страна се изчисляват от ляво на дясно.

– Цикълът `while` се изпълнява дотогава, докато условието (в случая: `b < 10`) е вярно. В Питон, подобно на C, всяка ненулева стойност е истина; нула е неистина. Също така, условието може да бъде символен низ или списък, всъщност каквато и да е редица (sequence). Всичко с ненулева дължина е истина, празните редици са неистина. Проверката, използвана в примера, е просто сравнение. Стандартните оператори за сравнение се изписват по същия начин, както и в C: `<` (по-малко), `>` (по-голямо), `==` (равно), `<=` (по-малко или равно), `>=` (по-голямо или равно) и `!=` (неравно).

– *Тялото* на цикъла е с *отстъп*: отстъпът е начина на Питон за групиране на операторите. Питон не предлага (засега!) интелигентно средство за редактиране на редовете, така че ще трябва да въвеждате табулация или интервал(и) за всеки ред с отстъп. В практиката ще подготвяте по-сложния вход за Питон с текстов редактор; повечето текстови редактори имат средство за автоматичен отстъп. Когато съставен оператор се въвежда интерактивно, той трябва да бъде последван от празен ред за да се укаже, че е приключил (тъй като граматичният анализатор не може да познае кога сте написали последния ред). Обърнете внимание, че всички редове, образуващи общ блок, трябва да бъдат с еднакъв отстъп.

– Операторът `print` изписва стойността на израза (или изразите), които са му подадени. Той се различава от простото изписване на израза който искате да покажете (както правихме по-горе в примерите с калкулатора) по начина по който се обработват символните низове[2] и съставните изрази. Символните низове се отпечатват без кавички и между елементите се вмъква интервал, така че можете добре да форматирате нещата. Например:

```
>>> i = 256*256
>>> print 'Стойността на i е', i
Стойността на i е 65536
```

Чрез оставена накрая запетайка се избягва преминаването на нов ред след изхода.

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Забележете, че интерпретаторът вмъква нов ред преди да е показал следващия си промпт, ако последният ред не е бил завършен.

[2] Както вече споменахме, операторът `print` е единствения начин да се покаже кирилица в текстовата конзола на Питон. *(бел. прев.)*

ГЛАВА
ЧЕТВЪРТА

Повече средства за управление на изпълнението

Освен току-що представения оператор `while`, Питон познава и обичайните оператори за управление на изпълнението, познати от други езици, но с някои особености.

4.1. `if`-конструкции

Може би най-познатият вид конструкции са `if`-конструкциите. Например:

```
>>> #  x = int(raw_input("Моля, въведете едно число: "))
>>> if x < 0:
...      x = 0
...      print 'Отрицателно, сменено на нула'
... elif x == 0:
...      print 'Нула'
... elif x == 1:
...      print 'Едно'
... else:
...      print 'Повече'
...
```

Може да има нула или повече `elif` части. Частта `else` е незадължителна. Ключовата дума '`elif`' е съкратено от 'else if', и е полезна за избягване на излишния отстъп. Поредицата `if` ... `elif` ... `elif` ... замества конструкциите *switch* или *case*, които намираме в други езици.

4.2. `for`-конструкции

Операторът `for` в Питон малко се различава от това, което може би сте използвали в C или Паскал. Вместо винаги да се итерира върху някаква аритметична прогресия на числа (като в Паскал), или да се дава на потребителя възможност да определя и стъпката на итерация, и условието за спиране (като в C), в Питон операторът `for` итерира върху елементите на каквато и да е редица (напр. списък или символен низ), в реда, по който те се явяват в редицата. Например (без намерение за каламбур):

```
>>> # Да измерим няколко символни низа:
... a = ['котка', 'прозорец', 'изхвърляне навън']
>>> for x in a:
...     print x, len(x)
...
котка 5
прозорец 8
изхвърляне навън 16
```

Не е безопасно да се променя редицата, докато се итерира върху нея в цикъл (това може да се случи само с променливите типове редици, т.е. списъците). Ако трябва да променяте списъка върху който итерирате, напр. за да дублирате избрани елементи, тогава е необходимо да итерирате върху

копие. Конвенцията за изрязване улеснява това частично:

```
>>> for x in a[:]: # прави копие чрез изрязване на целия списък
...     if len(x) > 8: a.insert(0, x)
...
>>> a
['изхвърляне навън', 'котка', 'прозорец', 'изхвърляне навън']
```

4.3. Функцията `range()`

Ако трябва да итерирате върху редица от числа, тогава идва на помощ вградената функция `range()`. Тя генерира списъци, съдържащи аритметични прогресии, напр.:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Подадената крайна точка никога не е част от генерирания списък; `range(10)` генерира списък от 10 стойности, съвпадащи с легалните индекси за елементите на редица с дължина 10. Възможно е да накарате диапазона да започва от друго число, или да определите различна стъпка на увеличение (дори отрицателна):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

За да итерирате върху индексите в редица, комбинирайте функциите `range()` и `len()` както следва:

```
>>> a = ['Когато', 'бях', 'овчарче', 'и', 'овците', 'пасях']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Когато
1 бях
2 овчарче
3 и
4 овците
5 пасях
```

4.4. Оператори `break` и `continue`, и клаузи `else` в циклите

Операторът `break`, както в C, излиза от най-вътрешния `for` или `while` цикъл.

Операторът `continue`, също взет назаем от C, продължава със следващата итерация в цикъла.

Циклите могат да имат клауза `else`; тя се изпълнява когато цикълът завърши чрез изчерпване на списъка (при `for`) или когато условието стане неистина (при `while`), но не и когато цикълът завърши с оператор `break`. Това е демонстрирано от следващия цикъл, който търси прости (неделими) числа:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, 'е равно на', x, '*', n/x
...            break
...     else:
...          print n, 'е просто число'
...
2 е просто число
3 е просто число
4 е равно на 2 * 2
5 е просто число
6 е равно на 2 * 3
7 е просто число
8 е равно на 2 * 4
9 е равно на 3 * 3
```

4.5. Операторът `pass`

Операторът `pass` не прави нищо. Той може да бъде използван, когато синтактически се очаква оператор, но програмата изисква бездействие. Например:

```
>>> while 1:
...       pass # Чакай за прекъсване от клавиатурата
...
```

4.6. Дефиниране на функции

Можем да създадем функция, която изписва числата на Фибоначи в произволна граница:

```
>>> def fib(n):    # изписва числата на Фибоначи до n
...     "Изписва числата на Фибоначи до n"
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Сега да извикаме току-що дефинираната функция:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Ключовата дума `def` въвежда *дефиниция* на функция. Тя трябва да бъде последвана от името на фукнцията и списък на формалните параметри в скоби. Операторите, образуващи тялото на функцията, започват от следващия ред и трябва да бъдат с отстъп. Първият оператор от тялото на функцията може да бъде символен низ. Този символен низ представлява документация за функцията, или *документационен символен низ* (docstring).

Съществуват средства, които използват документационните символни низове за да създадат автоматично печатна документация, или за да позволят на потребителя да се разхожда из кода. Добра практика е да включвате документационни символни низове в кода, който пишете, затова се опитайте да го превърнете в навик.

Изпълнението на функция въвежда нова символна таблица, предназначена за локалните променливи на функцията. По-точно, всички присвоявания на променливи в една функция запазват стойността в локалната символна таблица. Докато при обръщения към променливи се търси първо в локалната символна таблица, после в глобалната символна таблица, и накрая в таблицата на вградени имена. По този начин, във функцията не може директно да се присвои стойност на глобалните променливи (освен ако не се укаже оператор `global`), макар че могат да се правят обръщения към тях.

Актуалните параметри (аргументи) на едно извикване на функция се въвеждат в локалната символна таблица на извикваната функция, в момента, в който тя бива извикана. Така, аргументите се предават чрез *извикване по стойност* (където *стойността* винаги е *указател* към обект, а не самата стойност на обекта).[1] Когато една функция извиква друга функция, се създава нова локална символна таблица за това извикване.

Една дефиниция на функция въвежда името на функцията в текущата символна таблица. Стойността на името на фукнцията има тип, който се разпознава от интерпретатора като дефинирана от потребителя функция. Тази стойност може да бъде присвоявана на друго име, което после също може да бъде използвано като функция. Това служи като общ механизъм за преименуване:

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Може да възразите, че `fib` не е функция, а процедура. В Питон, както в C, процедурите са просто функции, които не връщат стойност. В действителност, технически казано, процедурите връщат стойност, макар и твърде безинтересна. Тази стойност се нарича `None` (това е вградено име). Обикновено изписването на стойността `None` се подтиска от интерпретатора, ако тя е единствената стойност, която ще бъде изведена. Можете да я видите, ако наистина държите на това:

```
>>> print fib(0)
None
```

Просто е да се напише функция, която връща списък от числата на Фибоначи, вместо да ги изписва.

```
>>> def fib2(n): # връща числата на Фибоначи до n
...     "Връща списък, съдържащ числата на Фибоначи до n"
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # виж по-долу
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # извикай я
>>> f100                # изпиши резултата
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Този пример, както обикновено, показва някои нови свойства на Питон:

- Операторът `return` връща изпълнението от функцията, предавайки стойност. `return` без израз като аргумент се използва за връщане от средата на процедура (стигането до края й също връща изпълнението от процедурата). В този случай се връща стойността `None`.
- Операторът `result.append(b)` извиква *метод* на обекта от тип списък. Методът е функция, която "принадлежи" на даден обект и се именува `обект.име_на_метод`, където `име_на_метод` е името на метода, който е дефиниран от типа на обекта. Различните типове дефинират различни методи. Методите на различни типове могат да имат еднакви имена, без да пораждат двусмисленост. (Възможно е да дефинирате ваши собствени обектни типове и методи, използвайки *класове*, както ще бъде обсъдено по-късно в това ръководство.) Методът `append()`, показан в този пример, е дефиниран за обектите от тип списък. Той добавя нов елемент в края на списъка. В този пример той е еквивалентен на '`result = result + [b]`', но е по-ефикасен.

[1] В действителност, *извикване по обектен указател* би било по-добро описание, тъй като ако се подаде променлив обект, извикващият би видял всички промени, които извикването е направил по указателя (напр., вмъкнати елементи в списък).

4.7. Повече за дефинирането на функции

Освен това е възможно да се дефинират функции с променлив брой аргументи. Има три форми, които могат да бъдат комбинирани.

4.7.1. Стойности на аргументите по подразбиране

Най-полезната форма е да се определи стойност по подразбиране на един или повече аргументи. Това създава функция, която може да бъде извиквана с по-малко аргументи, отколкото е дефинирана, например:

```
def ask_ok(prompt, retries=4, complaint='Да или не, моля!'):
    while 1:
        ok = raw_input(prompt)
        if ok in ('д', 'да', 'мда', 'ъхъ'): return 1
        if ok in ('н', 'не', 'тц'): return 0
        retries = retries - 1
        if retries < 0: raise IOError, 'инат потребител'
        print complaint
```

Тази функция може да бъде извиквана например така: `ask_ok('Наистина ли желаете да напуснете?')` или например така: `ask_ok('Да припокрия ли файла?', 2)`.

Стойностите по подразбиране се изчисляват по време на дефинирането на функцията в обсега на самата дефиниция, така че например:

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

ще отпечата 5.

Важно предупреждение: Стойностите по подразбиране се изчисляват само веднъж. Това поражда разлика, когато стойността по подразбиране е променлив обект като списък или речник (dictionary). Например, следващата функция събира аргументите, които са ѝ подадени в последователни извиквания:

```
def f(a, l = []):
    l.append(a)
    return l
print f(1)
print f(2)
print f(3)
```

Това ще отпечата

```
[1]
[1, 2]
[1, 2, 3]
```

Ако не желаете стойността по подразбиране да бъде споделяна между последователните извиквания, можете вместо това да напишете функцията така:

```
def f(a, l = None):
    if l is None:
        l = []
    l.append(a)
    return l
```

4.7.2. Аргументи с ключови думи

Също така, функциите могат да бъдат извиквани като се използват аргументи с ключови думи във формата '*ключова_дума = стойност*'. Например, следващата функция:

```
def parrot(voltage, state='корав', action='изгърми', type='норвежко синьо'):
    print "-- Този папагал няма да", action,
    print "ако му пуснеш", voltage, "волта."
    print "-- Прекрасна перушина в", type
    print "-- Той е", state, "!"
```

може да бъде извиквана по следните начини:

```
parrot(1000)
parrot(action = 'ИЗГЪРМИ', voltage = 1000000)
parrot('хиляда', state = 'гушнал букета')
parrot('милион', 'лишен от живот', 'скочи')
```

но следващите извиквания биха били невалидни:

```
parrot()                          # липсва задължителен аргумент
parrot(voltage=5.0, 'dead')     # аргумент с не-ключова дума следва ключова дума
parrot(110, voltage=220)          # дублирана стойност за аргумент
parrot(actor='Георги Парцалев')  # непозната ключова дума
```

Изобщо, списъкът с аргументите може да съдържа всякакви позиционни аргументи, следвани от всякакви аргументи с ключови думи, където ключовите думи са избрани от имената на формалните параметри. Не е важно дали даден формален параметър има стойност по подразбиране или не. Не може обаче един аргумент да получи стойност повече от веднъж. Имената на формалните параметри, съответни на позиционните аргументи, не могат да бъдат използвани като ключови думи в едно и също извикване. Ето един пример, който пропада заради това ограничение:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Когато последният формален параметър има формата ****име*, той получава речник, съдържащ всички аргументи с ключови думи, които не съответстват на формален параметър. Това може да бъде комбинирано с формален параметър с формата **име* (описан в следващата подглава), който получава комплект (tuple), съдържащ позиционните аргументи извън списъка на формалните параметри. (**име* трябва да се яви преди ****име*.) Например, ако дефинираме функция, подобна на тази:

```
def drinkshop(kind, *arguments, **keywords):
    print "-- Имате ли", kind, '?'
    print "-- Съжалявам, свършихме целия", kind
    for arg in arguments: print arg
    print '-'*40
    for kw in keywords.keys(): print kw, ':', keywords[kw]
```

Тя може да бъде извикана например така:

```
dinkshop('"Владимир"', "Това е много тъжно, господине.",
           "Това наистина е много, МНОГО тъжно, господине.",
           client='Боян Тончев',
           shopkeeper='Иван Сапунджиев',
           sketch='Случка в магазина за алкохол')
```

и разбира се, ще изведе:

```
-- Имате ли "Владимир" ?
-- Съжалявам, свършихме всичкия "Владимир"
Това е много тъжно, господине.
Това е много, МНОГО тъжно, господине.
----------------------------------------
client : Боян Тончев
shopkeeper : Иван Сапунджиев
sketch : Случка в магазина за алкохол
```

4.7.3. Произволни списъци с аргументи

Накрая, най-рядко използваната възможност е да се укаже, че функцията може да бъде извиквана с произволен брой аргументи. Тези аргументи ще бъдат увити в комплект (tuple). Преди произволния брой аргументи, могат да се явят нула или повече нормални аргументи.

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4. Ламбда форми

Поради всеобщо желание, в Питон са добавени няколко свойства, които обикновено се откриват във функционалните езици и Lisp. С ключовата дума `lambda` могат да се създават малки анонимни функции. Ето функция, която връща сумата на двата си аргумента: '`lambda a, b: a+b`'. Ламбда формите могат да се използват навсякъде, където се изискват обекти от тип функция. Синтактически, те са ограничени до единичен израз. Семантически те са просто синтактична глезотия, заместваща нормалните дефиниции на функции. Както и вложените дефиниции на функции, ламбда формите не могат да се обръщат към променливи в съдържащия ги обсег, но това може да бъде превъзмогнато чрез разумното използване на стойностите по подразбиране на аргументите, например:

```
def make_incrementor(n):
    return lambda x, incr=n: x+incr
```

4.7.5. Документационни символни низове

Очертават се конвенции за съдържанието и форматирането на документационните символни низове.

Първият ред винаги трябва да бъде кратко, стегнато резюме на предназначението на обекта. За краткост то не бива явно да назовава името на обекта или неговият тип, тъй като тези данни са достъпни посредством други способи (с изключение на случаите, когато името е глагол, описващ действието на функция). Този ред трябва да започва с главна буква и да свършва с точка.

Ако има повече редове в документационния символен низ, вторият ред трябва да бъде празен, нагледно разделящ резюмето от останалото описание. Следващите редове трябват да представляват един или повече параграфи, описващи конвенциите за извикване на обекта, страничните му ефекти, и прочие.

Граматичният анализатор на Питон не премахва отстъпа от многоредовите символни низове, така че средствата, които обработват документацията трябва (по желание) сами да се погрижат за това. За целта се използва следната конвенция. Първият непразен ред *след* първия ред от символния низ определя размера на отстъпа за целия документационен символен низ. (Не можем просто да използваме първия ред, тъй като по принцип в съседство с него са отварящите кавички на символния низ, и затова неговият отстъп не е очевиден в символния низ.) Тогава празното пространство, "равностойно" на този отстъп, се премахва от началото на всички редове в символния низ. Редове, които са с по-малък отстъп не бива да се появяват, но ако все пак се появят, цялото им начално празно пространство трябва да се премахне. Равностойността на празното пространство трябва да се проверява след разгръщането на табулациите (обикновено до 8 интервала).

Ето един пример за многоредов документационен символен низ.

```
>>> def my_function():
...     """Не прави нищо, но нека да я документираме.
...
...     Ама не, наистина нищо не прави.
...     """
...     pass
...
>>> print my_function.__doc__
Не прави нищо, но нека да я документираме.

    Ама не, наистина нищо не прави.
```

ГЛАВА
ПЕТА

Структури от данни

Тази глава описва по-подробно някои неща, които вече сте научили, като добавя и нови.

5.1. Повече за списъците

Типът данни списък (list) притежава още няколко метода. Ето всички методи на списъците:

append(x) Добавя елемент в края на списъка; еквивалентно на `a[len(a):] = [x]`.

extend(L) Разширява списъка като добавя всички елементи от дадения списък; еквивалентно на `a[len(a):] = L`.

insert(i, x) Вмъква елемент в дадената позиция. Първият аргумент е индекса на елемента, преди който се вмъква, така че `a.insert(0, x)` вмъква в началото на списъка, а `a.insert(len(a), x)` е еквивалентно на `a.append(x)`.

remove(x) Премахва първия елемент от списъка, чиято стойност е **x**. Грешка е, ако няма такъв елемент.

pop([i]) Премахва елемента, намиращ се на дадената позиция в списъка и го връща като резултат. Ако не е указан индекс, `a.pop()` връща последния елемент в списъка. Също така елементът се премахва от списъка.

index(x) Връща индекса на първия елемент от списъка, чиято стойност е **x**. Грешка е, ако няма такъв елемент.

count(x) Връща броя на срещанията на **x** в списъка.

sort() Сортира елементите в списъка, на място.

reverse() Обръща наопаки елементите в списъка, на място.

Един пример, който използва повечето от методите на списъка:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1. Използване на списъци като стекове

Методите на списъка правят много лесна употребата му като стек, където последният добавен елемент се извлича първи ("last-in, first out" – "последен вътре, първи вън"). За да добавите елемент на върха на стека, използвайте `append()`. За да извлечете елемент от върха на стека, използвайте `pop()` без явен индекс. Например:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. Използване на списъци като опашки

Също така, можете удобно да използвате списък като опашка, където първият добавен елемент се извлича първи ("first-in, first-out" – "първи вътре, първи вън"). За да добавите елемент в края на опашката, използвайте `append()`. За да извлечете елемент от началото на опашката, използвайте `pop()` с индекс `0`. Например:

```
>>> queue = ["Пешо", "Гошо", "Станко"]
>>> queue.append("Кольо")          # Кольо пристига
>>> queue.append("Биляна")         # Биляна пристига
>>> print queue.pop(0)
Пешо
>>> print queue.pop(0)
Гошо
>>> for i in queue: print i,
Станко Кольо Биляна
```

5.1.3. Функционални програмни средства

Съществуват три вградени функции, които са много полезни, когато се използват със списъци: `filter()`, `map()`, и `reduce()`.

'`filter(`*функция*`, `*редица*`)`' връща редица (от същия тип, ако е възможно), съдържаща тези елементи от редицата, за които *функция*`(`*елемент*`)` е вярно. Например, да изчислим няколко прости числа:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

'`map(`*функция*`, `*редица*`)`' извиква *функция*`(`*елемент*`)` за всеки от елементите на редицата и връща списък с върнатите стойности. Например, нека да изчислим няколко куба:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Може да бъде подадена повече от една редица. Функцията трябва да приема толкова аргументи, колкото са редиците. Тя се извиква със съответния елемент от всяка редица (или `None`, ако някоя редица е по-къса от другите). Ако вместо функция е подадена стойността `None`, тя се подменя с функция, която просто връща своите аргументи.

Комбинирайки тези два особени случая, виждаме, че '`map(None, `*списък1*`, `*списък2*`)`' е удобен начин за превръщане на чифт списъци в списък от чифтове. Например:

```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

'`reduce(`*функ*`, `*редица*`)`' връща единична стойност, получена от извикването на функцията с два аргумента *функ* с първите два елемента на редицата, после с резултата и следващия елемент и така нататък. Например, за да се изчисли сбора на числата от 1 до 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Ако в редицата има само един елемент, тогава се връща неговата стойност; ако редицата е празна, се предизвиква изключение (exception).

Може да бъде подаден трети аргумент, за да се укаже начална стойност. В този случай началната стойност се връща, ако редицата е празна, и функцията първо се прилага върху началната стойност и първия елемент от редицата, после на резултата и следващия елемент и така нататък. Например,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

5.2. Операторът `del`

Съществува начин за премахване на елемент от списък, подавайки индекса му, вместо стойност — операторът `del`. Той може да бъде използван също и за премахване на изрезки от списък (което по-рано правихме с присвояване на празен списък към изрезката). Например:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` може да бъде използван също и за изтриване на самите променливи:

```
>>> del a
```

Оттук нататък обръщането към `a` е грешка (поне докато не ѝ се присвои друга стойност). По-късно ще открием други употреби на `del`.

5.3. Комплекти и редици

Видяхме, че списъците и символните низове имат много общи свойства, например индексирането и операциите с изрязване. Те са два примера за типове данни "редица". Тъй като Питон е развиващ се език, могат да бъдат добавени други типове редици. Съществува и още един стандартен тип редица: *комплектът*[1].

Комплектът се състои от някакъв брой стойности, разделени със запетаи, например:

```
>>> t = 12345, 54321, 'привет!'
>>> t[0]
12345
>>> t
(12345, 54321, 'привет!')
>>> # Комплектите могат да бъдат влагани един в друг:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'привет!'), (1, 2, 3, 4, 5))
```

Както виждате, в изхода комплектите винаги са затворени в скоби, така че да се тълкуват правилно вложените комплекти. Те могат да бъдат въвеждани със или без заграждащите скоби, макар и често скобите да са, така или иначе, задължителни (ако комплектът е част от по-голям израз).

Комплектите имат широка употреба, например двойката координати (x, y), записите в база данни за служащите, и прочие. Комплектите, подобно на символните низове, са непроменливи: не е възможно да присвоявате стойност на индивидуален елемент от комплект (все пак можете да имитирате голяма част от този ефект чрез изрязване и сливане).

Особен проблем е създаването на комплекти, съдържащи 0 или 1 елемент: за това синтаксисът включва няколко допълнителни чалъма. Празни комплекти се създават с празна двойка скоби; комплект с един елемент се създава със запетая след стойността (не е достатъчно да затворите единичната стойност в скоби). Грозно е, но върши работа. Например:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- забележете запетаята в края
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Операторът `t = 12345, 54321, 'привет!'` е един пример за *пакетиране на комплект*: стойностите `12345`, `54321` и `'привет!'` са пакетирани заедно в един комплект. Обратната операция също е възможна, например:

```
>>> x, y, z = t
```

Това се нарича съответно *разпакетиране на комплект*. Разпакетирането на комплект изисква списъкът от променливи от лявата страна да притежава същия брой елементи, колкото е дължината на комплекта. Забележете, че многократното присвояване в действителност е просто комбинация от пакетиране и разпакетиране на комплект!

Понякога се налага използването на съответната операция върху списъци: *разпакетиране на списък*. Това се поддържа чрез затварянето на списъка от променливи в квадратни скоби.

```
>>> a = ['шкембе', 'чесън', 100, 1234]
>>> [a1, a2, a3, a4] = a
```

5.4. Речници

Друг полезен тип данни, вграден в Питон, е *речникът*. Речниците понякога се откриват в други езици като "асоциативни памети" или "асоциативни масиви". За разлика от типовете данни "редица",

[1] На англ. "tuple". Пуристите с образование по математика биха предпочели да наричат този тип "наредена n-торка". Признавам, че те имат своите основания, но въпреки това предпочитам да го предам като "комплект". *(бел. прев.)*

които са индексирани с някакъв диапазон от числа, речниците са индексирани по *ключове*, които могат да бъдат от всеки непроменлив тип. Символни низове и числа винаги могат да бъдат ключове. Комплекти могат да бъдат използвани като ключове само ако съдържат символни низове, числа, или комплекти. Не можете да използвате списъци като ключове, тъй като списъците могат да бъдат променяни на място, посредством метода им `append()`.

Най-добрият начин да се мисли за речниците е като за неподредено множество от двойки *ключ:стойност*, с изискването ключовете да са уникални (в рамките на един речник). Чифт големи скоби създава празен речник: `{}`. Разполагането на разделен със запетаи списък от двойки ключ:стойност в големите скоби добавя начални двойки ключ:стойност в речника. Това също е и начинът, по който речниците се извеждат на изхода.

Основните операции върху речника са съхраняване на стойност с някакъв ключ и извличане на стойността, подавайки ключа. Възможно е също да се премахва двойка ключ:стойност чрез `del`. Ако съхранявате, използвайки вече използван ключ, старата стойност, асоциирана с този ключ, ще бъде забравена. Грешка е, ако се извлича стойност използвайки несъществуващ ключ.

Методът `keys()` на обект от тип речник връща списък с всички използвани в речника ключове в случаен ред (ако го искате сортиран, просто приложете метода `sort()` върху списъка от ключове). За да проверите дали даден ключ е в речника, използвайте метода `has_key()` на речника.

Ето един кратък пример, който използва речник:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

5.5. Повече за условията

Условията, използвани в `while` и `if` конструкциите по-горе, могат да съдържат и други оператори, освен сравнения.

Операторите за сравнение `in` и `not in` проверяват дали дадена стойност се среща (или не се среща) в даден обект от тип редица. Операторите `is` и `is not` сравняват дали двата обекта действително са един и същ обект; това има значение само за променливите обекти като списъците. Всички оператори за сравнение имат един и същ приоритет, който е по-нисък от този на всички числови оператори.

Сравненията могат да бъдат навързвани: например, `a < b == c` проверява дали `a` е по-малко от `b` и освен това дали `b` е равно на `c`.

Сравненията могат да бъдат комбинирани с Булевите оператори `and` и `or`, а резултатът от сравнението (или от който и да е друг Булев израз) може да бъде отрицаван с `not`. Всички те отново имат по-нисък приоритет от операторите за сравнение; измежду тях `not` има най-висок приоритет, а `or` — най-малък, така че `A and not B or C` е еквивалентно на `(A and (not B)) or C`. Разбира се, могат да бъдат използвани скоби за да се изрази желаното съчетание.

Булевите оператори `and` и `or` са от т.нар. *пестеливи* оператори: техните аргументи се изчисляват от ляво на дясно, и изчисленията приключват в момента в който се определи резултата. Сиреч, ако `A` и `C` са истина, а `B` е неистина, `A and B and C` въобще не изчислява израза C. Изобщо, връщаната стойност на пестелив оператор, когато се използва като обикновена стойност, а не като Булева стойност, е последния изчислен аргумент.

Възможно е резултатът от сравнение или друг Булев израз да се присвои на променлива. Например,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Забележете, че в Питон, за разлика от C, вътре в изразите не може да се яви присвояване. Програмистите на C може да мърморят заради това, но така се избягват един цял клас проблеми, често срещани в програмите на C: въвеждане на = в израз, когато се е имало предвид ==.

5.6. Сравняване на редици и други типове

Обектите от тип редица могат да бъдат сравнявани с други обекти от същия тип редица. Сравнението използва *лексикографска* подредба: първо се сравняват първите два елемента, и ако те се различават, това определя резултата от сравнението; ако те са равни, сравняват се следващите два елемента, и така нататък, докато се изчерпи някоя от редиците. Ако двата елемента за сравнение са сами по себе си редици от един и същ тип, лексикографското сравнение се провежда рекурсивно. Ако всички елементи на две редици са равни, двете редици се считат за равни. Ако една редица е начална редица на друга, по-късата редица е по-малка. Лексикографската подредба на символните низове използва ASCII подредба за отделните символи. Ето няколко примера за сравнение между редици от един и същ тип:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)              == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Забележете, че е легално сравняването на обекти от различен тип. Резултатът е определим, а не случаен: типовете са подредени според името си. Така, списъкът (list) винаги е по-малък от символния низ (string), а символният низ винаги е по-малък от комплекта (tuple), и прочие. Смесените числови типове се сравняват според числовата им стойност, така че 0 е равно на 0.0, и прочие.[2]

[2] Правилата за сравняване на обекти от различен тип не са нещо, на което трябва да се разчита; те могат да бъдат променени в една бъдеща версия на езика.

ГЛАВА
ШЕСТА

Модули

Ако излезете от интерпретатора на Питон и влезете отново, дефинициите които сте направили (функции и променливи) се губят. Следователно, ако желаете да напишете малко по-дълга програма, по-добре е да използвате текстов редактор, за да подготвите входа за интерпретатора и да го стартирате с този файл като вход. Това е известно като създаване на *скрипт*. Когато програмата Ви порастне, може би ще пожелаете да я разделите на няколко файла за по-лесна поддръжка. Също така, може да пожелаете да използвате написана от Вас полезна функция в няколко програми, без да копирате дефиницията ѝ във всяка програма.

За да поддържа тези неща, Питон разполага с начин за поставяне на дефинициите във файл и използването им в скрипт или в интерактивна инстанция на интерпретатора. Такъв файл се нарича *модул*; дефинициите от даден модул могат да бъдат *импортирани* в други модули или в *главния* модул (сбора от променливи, към които имате достъп в даден скрипт, изпълняващ се на най-високо ниво и в режим на калкулатор).

Модулът е файл, съдържащ дефиниции и оператори на Питон. Файловото име е името на модула с добавен суфикс '.py'. В рамките на един модул, името му (като символен низ) е достъпно като стойност на глобалната променлива `__name__`. Например, използвайте любимия си текстов редактор за да създадете файл, наречен 'fibo.py' в текущата директория със следното съдържание:

```
# Модул за числата на Фибоначи

def fib(n):    # извежда числата на Фибоначи до n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # връща числата на Фибоначи до n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Сега влезте в интерпретатора на Питон и импортирайте този модул със следната команда:

```
>>> import fibo
```

Така имената на функциите, дефинирани във `fibo`, не се внасят направо в текущата символна таблица; внася се само името на модула `fibo`. Чрез името на модула получавате достъп до функциите:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Ако възнамерявате често да използвате дадена функция, можете да я присвоите на локално име:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. Повече за модулите

Освен дефиниции на функции, модулите могат да съдържат и изпълними оператори, които са предназначени да инициализират модула. Те се изпълняват само *първия* път, когато модулът е импортиран някъде.[1]

Всеки модул притежава своя собствена символна таблица, която се използва като глобална символна таблица от всички функции, дефинирани в модула. По този начин, авторът на модула може да използва глобални променливи в модула, без да се притеснява от евентуални конфликти с глобалните променливи на потребителя. От друга страна, ако знаете какво правите, можете да се доберете до глобалните променливи на даден модул със същата нотация, използвана и за обръщение към неговите функции — `име_на_модул.име_на_елемент`.

Модулите могат да импортират други модули. Обичайно е, но не задължително, всички оператори `import` да се разполагат в началото на модула (или скрипта). Имената на импортираните модули се разполагат в глобалната символна таблица на импортиращия модул.

Съществува вариант на оператора `import`, който импортира имената от даден модул направо в символната таблица на импортиращия модул. Например:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Това не внася името на модула, от който са импортирани имената, в локалната символна таблица (така че в този пример, името `fibo` не е дефинирано).

Дори съществува вариант, чрез който се импортират всички имена, които даден модул дефинира:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Така се импортират всички имена, с изключение на започващите с долна черта (_).

6.1.1. Път за търсене на модули

Когато се импортира модул с име `spam`, интерпретаторът търси файл с име '`spam.py`' в текущата директория. А сетне в списък от директории, определен от променливата $PYTHONPATH от обкръжението (environment). Тя има същия синтаксис, както и променливата $PATH от обвивката, сиреч списък от имена на директории. Когато $PYTHONPATH не е установена, или когато файлът не е открит там, търсенето продължава в път по подразбиране, който зависи от инсталацията; в UNIX, това обикновено е '`.:/usr/local/lib/python`'.

Всъщност модулите се търсят в списък от директории, даден от променливата `sys.path`, която се инициализира с директорията, съдържаща входния скрипт (или текущата директория), $PYTHONPATH и стойността по подразбиране, зависима от инсталацията. Това позволява на програми на Питон, които знаят какво вършат, да модифицират или подменят пътя за търсене на модули. Виж по-долу главата за Стандартните модули.

6.1.2. "Компилирани" файлове на Питон

Ако съществува файл, наречен '`spam.pyc`' в директория където се намира '`spam.py`', тогава се приема, че той вече съдържа "байт-компилирана" версия на модула `spam`. Целта е да се ускори времето за стартиране на кратки програми, използващи множество стандартни модули, ако съществува файл, наречен '`spam.pyc`' в директория където се намира '`spam.py`', тогава се приема, че той съдържа вече "байт-компилирана" версия на модула `spam`. Времето на последната промяна на версията на '`spam.py`',

[1] В действителност, дефинициите на функции също са "оператори", които се "изпълняват". Изпълнението внася името на функцията в глобалната символна таблица на модула.

използвана за да се създаде ‘spam.pyc’, е записано в ‘spam.pyc’. Ако времената не съвпадат, файлът се игнорира.

Обикновено няма нужда да правите нищо, за да създадете файла ‘spam.pyc’. Всеки път, когато ‘spam.py’ е компилиран успешно, се прави опит да се запише компилираната версия в ‘spam.pyc’. Не е грешка, ако този опит се провали. Ако поради някаква причина файлът не е напълно записан, получилият се файл ‘spam.pyc’ ще бъде разпознат като невалиден и, така, по-късно игнориран. Съдържанието на файла ‘spam.pyc’ е независимо от платформата, така че директория с модули на Питон може да бъде споделяна от машини с различни архитектури.

Няколко съвета за специалисти:

- Когато интерпретаторът на Питон е извикан с флага `-O` (това е буквата, а не цифрата), тогава се създава оптимизиран код, който се запазва в ‘.pyo’ файлове. Засега оптимизаторът не помага кой знае колко; той само премахва операторите `assert` и инструкциите `SET_LINENO`. Когато е използван `-O`, се оптимизира *всичкия* байткод; `.pyc` файловете се игнорират и `.py` файловете се компилират до оптимизиран байткод.
- Подаването на два флага `-O` към интерпретатора на Питон (`-OO`) ще накара компилатора на байткод да извърши оптимизации, които могат в някои редки случаи да доведат до неправилно функциониращи програми. Засега от байткода само се премахват документационните символни низове `__doc__`, което довежда до по-компактни ‘.pyo’ файлове. Тъй като някои програми могат да разчитат на наличието на тези документационни символни низове, трябва да използвате тази опция само ако знаете какво правите.
- Програмата въобще не върви по-бързо, когато се чете от ‘.pyc’ или ‘.pyo’ файл, вместо когато се чете от ‘.py’ файл; единственото по-бързо нещо при ‘.pyc’ или ‘.pyo’ файловете е скоростта, с която се зареждат.
- Когато даден скрипт се стартира с подаване на името му от командния ред, байткодът за този скрипт никога не се записва в ‘.pyc’ или ‘.pyo’ файл. Така времето за стартиране на един скрипт може да бъде намалено като се премести повечето от кода му в модул, оставяйки само малък стартиращ скрипт, който импортира този модул.
- Възможно е да съществува файл ‘spam.pyc’ (или ‘spam.pyo’, когато е използван `-O`), без в същата директория да съществува файл ‘spam.py’. Това може да служи за разпространение на библиотека от код на Питон във форма, относително трудна за дизасемблиране (reverse engineer).
- Модулът `compileall` може да създава ‘.pyc’ файлове (или ‘.pyo’ файлове, когато е използван `-O`) за всички модули в дадена директория.

6.2. Стандартни модули

Питон идва с библиотека от стандартни модули, описана в отделен документ, *Python Library Reference* (“Library Reference” от тук нататък). Някои модули са вградени в интерпретатора; те осигуряват достъп до операции, които не са част от същинския език, но са така или иначе вградени, дали заради производителност или за да осигурят достъп до примитивите на операционната система, като системни извиквания, например. Множеството от такива модули е въпрос на конфигурационен избор; например, модулът `amoeba` се обезпечава само върху системи, които по някакъв начин поддържат Amoeba примитиви. Един особен модул заслужава малко повече внимание: `sys`, който е вграден във всеки интерпретатор на Питон. Променливите `sys.ps1` и `sys.ps2` дефинират символните низове, използвани като вторични и първични промптове:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Язък!'
Язък!
C>
```

Тези две променливи са дефинирани само ако интерпретаторът е в интерактивен режим.

Променливата `sys.path` е списък от символни низове, който определя пътя за търсене на модули от интерпретатора. Той се инициализира с пътя по подразбиране, взет от променливата $PYTHONPATH на обкръжението, или от вградена стойност по подразбиране, ако $PYTHONPATH не е дефинирана. Можете да я променяте посредством обикновените операции върху списък, например:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. Функцията `dir()`

Вградената функция `dir()` се използва, за да се открие какви имена дефинира даден модул. Тя връща сортиран списък от символни низове.

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
```

Без аргументи, `dir()` прави списък на имената, които понастоящем сте дефинирали:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Забележете, че тя прави списък на всички видове имена: променливи, модули, функции, и прочие.

`dir()` не прави списък на имената на вградените функции и променливи. Ако желаете техния списък, те са дефинирани в стандартния модул `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
```

6.4. Пакети

Пакетите са начина на Питон за структуриране на имената на модулите, използвайки "точкувани имена на модули". Например, името на модул `A.B` обозначава подмодул с име '`B`' в пакет с име '`A`'. Точно както използването на модули предпазва авторите на различни модули от безпокойството помежду им за имената на глобалните променливи, така използването на точкувани имена на модули предпазва авторите на многомодулни пакети като NumPy или Python Imaging Library от безпокойството помежду им относно имената на модулите.

Да предположим, че Вие искате да проектирате колекция от модули ("пакет") за общо управление на звукови файлове и звукови данни. Съществуват много различни звукови файлови формати (обикновено разпознавани по разширението им, например '.wav', '.aiff', '.au'), така че може да се нало-

жи да създадете и поддържате растяща колекция от модули за конвертиране между разнообразните файлови формати. Също така, съществуват много различни операции, които може би ще желаете да изпълнявате върху звуковите данни (например миксиране, добавяне на ехо, прилагане на еквалайзерна функция, създаване на изкуствен стерео ефект), така че като добавка ще пишете безкраен поток от модули за прилагане на тези операции. Ето една възможна структура на Вашия пакет (изразен с термините на йерархична файлова система):

```
Sound/                          Пакет от най-високо ниво
      __init__.py               Инициализация на звуковия пакет
      Formats/                  Подпакет за конвертиране на файлови формати
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  Подпакет за звукови ефекти
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  Подпакет за филтри
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Файловете '__init__.py' са необходими за да накарат Питон да се отнася към директориите като съдържащи пакети. Това е направено за да се предотвратят случите, когато директориите с общоприето име, например 'string', случайно скриват валидни модули, които се появяват по-късно в пътя за търсене на модули. В най-простия случай, '__init__.py' може да бъде просто празен файл, но той може и да изпълнява инициализиращ код за пакета, или да установява променливата __all__, описана по-долу.

Потребителите на пакета могат да импортират отделни модули от пакета, например:

```
import Sound.Effects.echo
```

Така се зарежда подмодула Sound.Effects.echo. Към него трябва да се обръщате с пълното му име, например:

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Друг начин за импортиране на подмодул е:

```
from Sound.Effects import echo
```

Зареден по този начин, подмодулът echo е достъпен без префикса на пакета му, така че той може да бъде използван както следва:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Още един вариант за непосредствено импортиране на желаната функция или променлива:

```
from Sound.Effects.echo import echofilter
```

Още веднъж, така се зарежда подмодула echo, но по такъв начин, че неговата функция echofilter() е непосредствено достъпна:

```
echofilter(input, output, delay=0.7, atten=4)
```

Забележете, че когато се използва `from` *пакет* `import` *елемент*, елементът може да бъде или подмодул (или подпакет) на пакета, или някакво друго дефинирано в пакета име, като функция, клас или променлива. Операторът `import` първо проверява дали елемента е дефиниран в пакета; ако не е, приема, че е модул и пробва да го зареди. Ако не успее да го открие, се предизвиква изключението `ImportError`.

Обратно, когато се използва синтаксис подобен на `import` *елемент.поделемент.подподелемент*, всеки елемент, освен последния, трябва да бъде пакет. Последният елемент може да бъде модул или пакет, но не може да бъде клас, функция или променлива, дефинирани в предходния елемент.

6.4.1. Импортиране на * от пакет

Какво се случва, когато потребителят напише `from Sound.Effects import *`? В идеалния случай, някой би си помислил, че така по някакъв начин се излиза във файловата система, откриват се наличните в пакета модули, и всички те се импортират. За нещастие, тази операция не работи много добре върху Mac и Windows платформи, където файловата система не винаги има точна информация за регистъра (т.е. главните и малките букви) на файловите имена. На тези платформи няма сигурен начин да се разбере дали файлът '`ECHO.PY`' трябва да бъде импортиран като модул `echo`, `Echo` или `ECHO`. (Например, Windows 95 има смущаващата практика да показва всички файлови имена с главна първа буква.) Ограничението на DOS за файлово име от 8+3 добавя още един интересен проблем за дългите имена на модули.

Единственото решение е авторът на пакета да даде изричен индекс на пакета. Импортиращият оператор използва следната конвенция: ако кодът във файла '`__init__.py`' на даден пакет дефинира списък с име `__all__`, се приема че той представлява списък на имената на модулите, които трябва да бъдат импортирани, когато се срещне `from` *пакет* `import *`. Авторът на пакета носи отговорността да държи този списък актуален, когато се пуска нова версия на пакета. Авторите на пакети могат също така да решат, че няма да поддържат този списък, ако не виждат смисъл за импротиране на * от техния пакет. Например, файлът '`Sounds/Effects/__init__.py`' би могъл да съдържа следния код:

```
__all__ = ["echo", "surround", "reverse"]
```

Това би означавало, че `from Sound.Effects import *` би импортирал трите посочени подмодула от пакета `Sound`.

Ако `__all__` не е дефиниран, операторът `from Sound.Effects import *` *не* импортира всички подмодули на пакета `Sound.Effects` в текущото пространство на имената (namespace); той само осигурява импортирането на пакета `Sound.Effects` (стартирайки неговия инициализиращ код, '`__init__.py`', ако е възможно) и после импортира имената, които са дефинирани в пакета. Това включва всички имена, дефинирани (и изрично заредени подмодули) от '`__init__.py`'. Също така, това включва всякакви подмодули на пакета, които са били изрично заредени от предишни импортиращи оператори, например:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

В този пример, модулите echo и surround се импортират в текущото пространство на имената, защото са дефинирани в пакета `Sound.Effects`, когато операторът `from...import` е изпълнен. (Това също работи когато е дефиниран `__all__`.)

Забележете, че като цяло е кофти импортирането на * от модул или пакет, тъй като често води до трудно четим код. Обаче, то е добре дошло ако го използвате за да си спестите набиране в интерактивните сесии. Също така, определени модули са проектирани да експортират само имена, които следват определен модел.

Помнете, че няма нищо лошо в това да се използва `from Пакет import определен_подмодул`! В действителност, това е препоръчителният запис, освен ако импортиращият модул не трябва да използва подмодули със същото име, но от други пакети.

6.4.2. Вътрешнопакетни обръщения

Подмодулите често имат нужда да се обръщат един към друг. Например, модулът `surround` може да използва модула `echo`. В действителност, подобни обръщения са толкова обичайни, че операторът `import` първо търси в съдържащия пакет, преди да прегледа стандартния път за търсене. Така, модулът surround може просто да използва `import echo` или `from echo import echofilter`. Ако импортираният модул не е открит в текущия пакет (пакетът, на който текущия модул е подмодул), операторът `import` търси за модул с даденото име от най-високо ниво.

Когато пакетите са структурирани в подпакети (както пакетът `Sound` в нашия пример), няма кратък начин за обръщение към подмодулите на съседните пакети — трябва да бъде използвано пълното име на подпакета. Например, ако модулът `Sound.Filters.vocoder` трябва да използва модула `echo` от пакета `Sound.Effects`, той може да използва `from Sound.Effects import echo`.

ГЛАВА
СЕДМА

Вход и изход

Съществуват няколко начина да се представи изхода на една програма: данните могат да бъдат изведени в подходяща за четене от човек форма, или да бъдат записани във файл за бъдеща употреба. Тази глава ще обсъди някои от възможностите.

7.1. По-гиздаво форматиране на изхода

Дотук срещнахме два начина за извеждане на стойности: *изразите* и оператора `print`. (Трети начин е да се използването на метода `write()` на файловите обекти — към файла на стандартния изход можете да се обръщате чрез `sys.stdout`. Вижте Library Reference за повече информация по този въпрос.)

Често бихте желали повече контрол върху форматирането на вашия изход, вместо простото извеждане на стойности, разделени с интервал. Съществуват два начина да форматирате изхода. Първият е сами да изпълните цялото управление на символните низове — чрез изрязване и съединяване на символни низове можете да създадете всеки изглед, който пожелаете. Стандартният модул `string` съдържа няколко полезни операции за допълване на символни низове до даден размер на колона; това ще бъде обсъдено накратко. Вторият начин е да използвате оператора `%` със символен низ като ляв аргумент. Операторът `%` интерпретира левия аргумент като форматиращ символен низ много подобен на `sprintf()` в C, който да бъде приложен към десния аргумент, и връща символния низ, получен от тази форматираща операция.

Разбира се, остава един въпрос — как ще превръщате стойности в символни низове? За щастие, Питон разполага с начин за превръщане на всяка стойност в символен низ: предайте я на функцията `repr()`, или просто изпишете стойността между обратни кавички (“). Няколко примера:

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'Стойността на x е ' + `x` + ', а на y е ' + `y` + '...'
>>> print s
Стойността на x е 31.4, а на y е 40000...
>>> # Обратните кавички работят не само върху числа, но и върху други типове:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.4, 40000]'
>>> # Превръщането на символен низ добавя кавички на символен низ и обратно наклонени черти:
... hello = 'hello, world\n'
>>> hellos = `hello`
>>> print hellos
'hello, world\012'
>>> # Аргументът на обърнатите кавички може да бъде комплект:
... `x, y, ('spam', 'eggs')`
"(31.4, 40000, ('spam', 'eggs'))"
```

Ето два начина да се изведе таблица с квадрати и кубове:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(`x`, 2), string.rjust(`x*x`, 3),
...     # Забележете останалата последна запетая на горния ред
...     print string.rjust(`x*x*x`, 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Забележете, че е добавен по един празен интервал между всяка колона, поради начина на работа на `print`, който винаги добавя интервали между аргументите си.)

Този пример демонстрира функцията `string.rjust()`, която подравнява от дясно даден символен низ в поле с дадена ширина, допълвайки го с интервали отляво. Съществуват подобни функции `string.ljust()` и `string.center()`. Тези функции не извеждат нищо, те просто връщат нов символен низ. Ако входният символен низ е твърде дълъг, те не го скъсяват, а го връщат непроменен. Това ще объркa вашия колонен изглед, но обикновено е за предпочитане пред алтернативата да се заблудите относно някоя стойност. (Ако наистина желаете скъсяване, винаги можете да добавите изрязваща операция, подобна на '`string.ljust(x, n)[0:n]`'.)

Съществува друга функция, `string.zfill()`, която добавя числов символен низ с нули отляво. Тя разпознава знаците за плюс и минус:

```
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

Използването на оператора `%` може да изглежда по следния начин:

```
>>> import math
>>> print 'Стойността на Пи е приблизително %5.3f.' % math.pi
Стойността на Пи е приблизително 3.142.
```

Ако има повече от един формат в символния низ, тогава подавате комплект като десен операнд, например:

```
>>> table = {'Светла': 758211, 'Боян': 744903, 'Пешо': 52230440}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Светла     ==>     758211
Пешо       ==>   52230440
Боян       ==>     744903
```

Повечето формати работят точно както в C и изискват от Вас да подадете точен тип; ако не го направите обаче, ще получите изключение, а не дъмп на ядрото. Форматът `%s` е по-разхлабен: ако съответният му аргумент не е обект от символен низ, той се конвертира до символен низ чрез вградената функция `str()`. Поддържа се използването на `*` за да се подаде размера или точността като отделен (целочислен) аргумент. Не се поддържат C форматите `%n` и `%p`.

Ако имате наистина дълъг форматиращ символен низ, който не искате да разделяте, би било чудесно ако можете да указвате форматирането на променливите по име, вместо по позиция. Това може да се постигне чрез едно разширение на C форматите с формата `%(име)формат`, сиреч:

```
>>> table = {'Светла': 758211, 'Боян': 744903, 'Пешо': 52230440}
>>> print 'Боян: %(Боян)d; Светла: %(Светла)d; Пешо: %(Пешо)d' % table
Боян: 744903; Светла: 758211; Пешо: 52230440
```

Това е особено полезно в комбинация с новата вградена функция `vars()`, която връща речник, съдържащ всички локални променливи.

7.2. Четене и писане на файлове

`open()` връща файлов обект object, и най-често се използва с два аргумента: '`open(`*файлово_име*`, `*режим*`)`'.

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Първият аргумент е символен низ, съдържащ файловото име. Вторият аргумент е друг символен низ, съдържащ няколко знака, описващи начина, по който файлът ще бъде използван. *режим* може да бъде `'r'`, което значи че файлът ще бъде само за четене[1], `'w'` е само за писане[2] (съществуващ файл със същото име ще бъде изтрит), и `'a'` отваря файла за добавяне[3]. Всички данни, писани във файла, автоматично се добавят в края. `'r+'` отваря файла и за четене и за писане. Аргументът *режим* е незадължителен; ако `'r'` е пропуснат, той се подразбира.

Върху Windows и Macintosh, прибавянето на `'b'` в режима отваря файла в двоичен[4] режим, така че съществуват също режими като `'rb'`, `'wb'`, и `'r+b'`. Windows прави разлика между текстови и двоични файлове; знаците за "край на ред" в текстовите файлове автоматично леко се променят, когато се четат или пишат данни. Това изменение "зад кулисите" на данните от файла е хубаво за ASCII текстови файлове, но ще повреди двоичните данни като тези в JPEG или '.EXE' файловете. Бъдете много внимателни при използването на двоичен режим, когато четете или пишете в такива файлове. (Забележете, че точната семантика на текстовия режим върху Macintosh зависи от лежащата отдолу C библиотека, която се използва.)

7.2.1. Методи на файловите обекти

За останалите примери в тази глава ще се приема, че вече е създаден файлов обект, наречен `f`.

За да четете съдържанието на файл, извикайте `f.read(`*размер*`)`, така се прочита някакво количество данни, които се връщат като символен низ. *размер* е незадължителен числов аргумент. Когато *размер* е пропуснат или е отрицателен, ще бъде прочетено и върнато цялото съдържание на файла; Ваш проблем е, ако файлът е два пъти по-голям от паметта на машината Ви. В противен

[1] от англ. read *(бел. прев.)*
[2] от англ. write *(бел. прев.)*
[3] от англ. append *(бел. прев.)*
[4] от англ. binary *(бел. прев.)*

случай ще се прочетат и върнат най-много *размер* байта. Ако е достигнат края на файла, `f.read()` ще върне празен символен низ (″).

```
>>> f.read()
'Това е целият файл.\012'
>>> f.read()
''
```

`f.readline()` прочита един ред от файла; знакът за нов ред (`\n`) е оставен на края на символния низ, и е изпуснат на посления ред само ако файлът не завършва със знак за нов ред. Това прави върнатата стойност недвусмислена. Ако `f.readline()` върне празен символен низ, тогава е бил достигнат края на файла. А празен ред се представя чрез `'\n'`, символен низ съдържащ само нов ред.

```
>>> f.readline()
'Това е първият ред от файла.\012'
>>> f.readline()
'Вторият ред от файла\012'
>>> f.readline()
''
```

`f.readlines()` използва `f.readline()` многократно, и връща списък, съдържащ всички редове от данни във файла.

```
>>> f.readlines()
['Това е първият ред от файла.\012', 'Вторият ред от файла\012']
```

`f.write(`*символен_низ*`)` записва съдържанието на *символен_низ* във файла, връщайки `None`.

```
>>> f.write('Това е проба\n')
```

`f.tell()` връща цяло число, което дава текущата позиция във файла на файловия обект, измерена в байтове от началото на файла. За да промените позицията на файловия обект, използвайте ‘`f.seek(`*отместване*`, `*от_кое*`)`’. Позицията е изчислена от добавянето на *отместване* към една отправна точка; отправната точка е избрана от аргумента *от_кое*. При стойност 0 на *от_кое* се измерва от началото на файла, при 1 се използва текущата файлова позиция, и при 2 се използва края на файла като отправна точка.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Отиди на 5-тия байт във файла
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Отиди на 3-тия байт преди края
>>> f.read(1)
'd'
```

Когато свършите работата си с даден файл, извикайте `f.close()`, за да го затворите и да освободите всички системни ресурси, заети от отворения файл. След извикването на `f.close()`, всички опити да се използва файловия обект автоматично пропадат.

```
>>> f.close()
>>> f.read()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Файловите обекти имат няколко допълнителни метода като `isatty()` и `truncate()`, които са по-рядко използвани. Консултирайте се с Library Reference за пълно описание на файловите обекти.

7.2.2. Модулът `pickle`

Символните низове могат лесно да бъдат записвани и четени от файл. Числата изискват повече усилие, тъй като методът `read()` връща само символни низове, които би трябвало да бъдат предавани на функции като `string.atoi()`, която взима символен низ като `'123'` и връща числовата му стойност 123. Обаче, ако искате да запазите по-сложни типове данни като списъци, речници, или инстанции на класове, нещата стават много по-сложни.

Вместо да се налага потребителите постоянно да пишат и тестват код за запазване на сложни типове данни, Питон предлага стандартен модул, наречен `pickle`. Това е удивителен модул, който взима почти всеки обект на Питон (дори някои форми на код на Питон!), и го превръща в символно представяне; този процес се нарича *мариноване*[5]. Реконструирането на обект от символното му представяне се нарича *размариноване*. Символният низ, представящ обекта между маринованието и размаринованието, може да бъде запазен във файл или данни, или изпратен през мрежова връзка към някоя отдалечена машина.

Ако имате обект `x`, и файлов обект `f`, който е бил отворен за писане, най-простият начин да мариновате обекта отнема само един ред код:

```
pickle.dump(x, f)
```

За да размариновате обекта отново, ако `f` е файлов обект отворен за писане:

```
x = pickle.load(f)
```

(Съществуват и други варианти, които се използват при маринованието на много обекти, или когато не желаете да записвате маринованите данни във файл. Консултирайте се с пълната документация на `pickle` в Library Reference.)

`pickle` е стандартният начин обектите на Питон да могат да бъдат запазвани и използвани отново от други програми или от друго изпълнение на същата програма. Техническият термин за това е *упорит* обект. Понеже `pickle` е толкова широко използван, много автори, които пишат разширения за Питон, се стараят да осигурят правилното мариноване и размариноване на новите типове данни (напр. матрици).

[5] на англ. pickling *(бел. прев.)*

ГЛАВА
ОСМА

Грешки и изключения

Досега не е обръщано много внимание на съобщенията за грешки, но може би сте видели някои от тях, ако сте опитвали примерите. Съществуват (поне) два различни вида грешки: *синтактични грешки (syntax errors)* и *изключения (exceptions)*.

8.1. Синтактични грешки

Синтактичните грешки, познати още като граматични грешки, са може би най-честият вид проблеми, които срещате, докато все още изучавате Питон:

```
>>> while 1 print 'Hello world'
  File "<stdin>", line 1
    while 1 print 'Hello world'
                ^
SyntaxError: invalid syntax
```

Граматичният анализатор повтаря засегнатия ред и изобрзява малка стрелка, сочеща към най-ранната точка в реда, където е открита грешката. Грешката е причинена от (или поне открита до) знака, *предшестващ* стрелката — в примера, грешката е открита в ключовата дума `print`, тъй като преди нея липсват двете точки (‘`:`’). Изведени са файловото име и номера на реда, така че да знаете къде да погледнете в случай, че изхода идва от скрипт.

8.2. Изключения

Дори когато даден оператор или израз е синтактически верен, той може да предизвика грешка, когато се направи опит да се изпълни. Грешките, открити по време на изпълнение се наричат *изключения* и не са безусловно фатални: скоро ще разберете как да ги управлявате в програмите си на Питон. Повечето изключения не се обработват от програмите, така че резултатът се показва в съобщения за грешка:

```
>>> 10 * (1/0)
Traceback (innermost last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (innermost last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (innermost last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

Последният ред от съобщението за грешка посочва какво се е случило. Изключенията биват от различен тип, и типът е изведен като част от съобщението: типовете в примера са `ZeroDivisionError`, `NameError` и `TypeError`. Символният низ, изведен като тип на изключението, е вграденото име на появилото се изключение. Това е вярно за всички вградени изключения, но не е непременно вярно за дефинираните от потребителя изключения (макар че е една полезна конвенция). Стандартните имена на изключения са вградени идентификатори (не са запазени ключови

думи).

Останалата част от реда е подробност, чието тълкуване зависи от типа на изключението; нейното значение е зависимо от типа на изкючението.

Предходната част от съобщението за грешка показва контекста, в който се е случило изключението, във формата на обратно трасиране на стека. Като цяло тя съдържа обратно трасиране на стека, изброявайки редовете от изходния код; тя обаче няма да покаже редовете, прочетени от стандартния вход.

Python Library Reference съдържа списък на вградените изключения и техните значения.

8.3. Обработка на изключения

Възможно е да пишете програми, които да обработват избрани изключения. Вижте следващия пример, който подканя потребителя за вход, докато не въведе валидно цяло число, но позволява на потребителя да прекъсне програмата (чрез `Control-C` или каквото там поддържа операционната система). Забележете, че причиненото от потребителя прекъсване се сигнализира чрез предизвикването на изключение `KeyboardInterrupt`.

```
>>> while 1:
...     try:
...         x = int(raw_input("Моля, въведете едно число: "))
...         break
...     except ValueError:
...         print "Опа! Това не беше валидно число. Опитайте отново..."
...
```

Операторът `try` работи както следва.

- Първо се изпълнява *try клаузата* (оператор(ите) между ключовите думи `try` и `except`).
- Ако не възникне изключение, *except клаузата* се пропуска и изпълнението на оператора `try` приключва.
- Ако възникне изключение по време на изпълнението на try клаузата, останалата част от клаузата се пропуска. Тогава ако неговият тип съвпада с изключението, назовано след ключовата дума `except`, останалата част от try клаузата се пропуска, клаузата except се изпълнява, и после изпълнението продължава след оператора `try`.
- Ако възникне изключение, което не съвпада с изключението, назовано в except клаузата, то се подава на по-външните `try` оператори; ако не се намери кой да го обработи, това е *необработено изключение* и изпълнението спира със съобщение подобно на показаните по-горе.

Един оператор `try` може да притежава повече от една except клауза, за да определи различни групи код за обработка на различните изключения. Поне една от групите код за обработка ще се изпълни. Групите за обработка могат да обработват само изключенията, които са се появили в съответната try клауза, но не в другите групи за обработка от същия оператор `try`. Една except клауза може да назове няколко изключения като разделен със запетаи списък, например:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Последната except клауза може да пропусне имената на изключенията, за да важи за всички. Използвайте това с крайна предпазливост, тъй като по този начин е лесно да маскирате реална програмна грешка! Можете да го използвате също за да изведете съобщение за грешка и после отново да предизвикате изключението (давайки възможност на извикващия код също да обработи изключението):

```
import string, sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Конструкцията `try ... except` притежава една незадължителна *else клауза*, която трябва да е след всички except клаузи. Полезно е там да се разполага код, който трябва да се изпълни ако try клаузата не предизвика изключение. Например:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'не мога да отворя', arg
    else:
        print arg, 'има', len(f.readlines()), 'реда'
        f.close()
```

Когато възникне изключение, то може да притежава допълнителна стойност, известна още като *аргумент* на изключението. Наличието и типът на аргумента зависят от типа на изключението. За типове изключения, притежаващи аргумент, except клаузата може да определи една променлива след името (или списъка) на изключението(-ята) за да получи стойността на аргумента, както следва:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'името', x, 'не е дефинирано'
...
name spam undefined
```

Ако дадено изключение притежава аргумент, той е изведен в последната част ("подробност") на съобщението за необработени изключения.

Групата код за обработка на изключение обработва изключенията не само когато се появят непосредствено в try клаузата, но също и ако се появят вътре в извикваните функции (дори непряко) в try клаузата. Например:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Обработка на грешка по време на изпълнение:', detail
...
Обработка на грешка по време на изпълнение: integer division or modulo
```

8.4. Предизвикване на изключения

Операторът `raise` позволява на програмиста да направи така, че да се появи определено изключение. Например:

```
>>> raise NameError, 'HiThere'
Traceback (innermost last):
  File "<stdin>", line 1
NameError: HiThere
```

Първият аргумент на `raise` назовава изключението, което следва да бъде предизвикано. Незадължителният втори аргумент определя аргумента на изключението.

8.5. Дефинирани от потребителя изключения

Програмите могат да притежават свои собствени изключения, чрез присвояване на символен низ на променлива или чрез създаване на нов клас изключения. Например:

```
>>> class MyError:
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return `self.value`
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'Появи се моето изключение, стойност:', e.value
...
Появи се моето изключение, стойност: 4
>>> raise MyError, 1
Traceback (innermost last):
  File "<stdin>", line 1
__main__.MyError: 1
```

Такива изключения се използват от много стандартни модули за да докладват грешки, които могат да се появят в дефинираните от тях функции.

Повече информация за класовете е представена в глава 9, “Класове”.

8.6. Дефиниране на почистващи действия

Операторът `try` притежава още една незадължителна клауза, която е предназначена да дефинира почистващи действия, които трябва да се изпълнят при всички положения. Например:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Сбогом, свят!'
...
Сбогом, свят!
Traceback (innermost last):
  File "<stdin>", line 2
KeyboardInterrupt
```

Дадена *finally клауза* се изпълнява независимо дали се е появило изключение в try клаузата, или не. Когато се е появило изключение, то се предизвиква отново след изпълнението на клаузата finally. Също така, клаузата finally се изпълнява “на изхода”, когато операторът `try` е напуснат чрез оператор `break` или `return`.

Даден оператор `try` трябва да притежава или една или повече клаузи except, или една клауза finally, но не и двете заедно.

ГЛАВА
ДЕВЕТА

Класове

С минимално количество нов синтаксис и семантика, механизмът на Питон за класове (classes) добавя класове към езика. Той е смес между механизмите за класове, намиращи се в C++ и Modula-3. Тъй както и при модулите, класовете в Питон не слагат безусловна бариера между дефиницията и потребителя, а по-скоро разчитат на учтивостта на потребителя да не "проникне с взлом в дефиницията". Най-важните свойства на класовете са запазени в пълната си мощ, тоест: механизмът за наследяване (inheritance) на класове позволява няколко базови класа, полученият клас може да припокрие (override) всеки метод от своя базов клас или класове, даден метод може да извиква метод на базов клас със същото име. Обектите могат да съдържат произволно количество частни (private) данни.

В терминологията на C++, всички членове на класа (включително членовете-данни[1]) са *публични (public)*, и всички членове-функции[2] са *виртуални (virtual)*. Не съществуват специални конструктори или деструктори. Както в Modula-3, не съществуват преки начини за обръщане към членовете на обекта от методите му: методът е деклариран с експлицитен първи аргумент, представляващ обекта, който се предоставя имплицитно с извикването. Както в Smalltalk, самите класове са обекти, макар и в по-широк смисъл на думата: в Питон всички типове данни са обекти. Това предоставя семантика за импортиране и преименуване. Но точно както в C++ или Modula-3, вградените типове не могат да се използват като базови класове за разширение от потребителя. Също така, както в C++ и за разлика от Modula-3, повечето вградени оператори със специален синтаксис (аритметични оператори, подписване и прочие) могат да бъдат предефинирани за инстанциите на класове.

9.1. Няколко думи за терминологията

Поради липсата на общоприета терминология, с която да се говори за класовете, ще употребявам от време на време термини от Smalltalk и C++. (Бих могъл да употребявам и термините на Modula-3, тъй като обектно-ориентираната семантика на този език е по-близка до тази на Питон, отколкото C++, но мисля, че малцина читатели въобще са чували за нея.)

Също така, трябва да предупредя, че тук съществува терминологична клопка за обектно-ориентираните читатели: в Питон, думата "обект" не значи задължително инстанция на клас. Подобно на C++ и Modula-3, и за разлика от Smalltalk, в Питон не всички типове са класове: основните вградени типове като целите числа и списъците не са класове, и дори не са класове малко по-екзотичните типове като файловете. Обаче, *всички* типове в Питон споделят някаква частица обща семантика, която се описва най-добре с думата "обект".

Обектите имат индивидуалност, и няколко различни имена (в различни обсеги) могат да бъдат свързвани към един и същ обект. В другите езици това е познато като псевдонимизиране (aliasing). Обикновено на пръв поглед то не е добре дошло в Питон. Обаче псевдонимизирането притежава един (нарочно търсен!) ефект върху семантиката на кода на Питон, повличайки след себе си променливите обекти като списъци, речници, и повечето типове, представящи някакви единици извън програмата (файлове, прозорци, и прочие). Това обикновено е предимство за програмата, тъй като псевдонимите в някои отношения се държат като указатели. Например, предаването на обект е евтино, тъй като на ниво реализация се предава само указател. И ако дадена функция измени обекта, който ѝ е подаден като аргумент, извикващият ще види промяната — това премахва необходимостта от два различни механизма за предаване на параметри както в Паскал.

[1] наричани по-често "полета". *(бел. прев.)*
[2] наричани по-често "методи". *(бел. прев.)*

9.2. Обсеги и пространства на имената в Питон

Преди да се запознаем с класовете, трябва първо да Ви кажа нещо за правилата за обсег на Питон. Дефинициите на класове правят няколко изкусни трика с пространствата на имената, и трябва да знаете как работят обсегите и пространствата на имената, за да разберете напълно какво става. Между другото, познанията върху този въпрос са полезни за всеки напреднал програмист на Питон.

Нека започнем с няколко дефиниции.

Пространство на имената (name space) е една асоциация на имена към обекти. Повечето пространства на имената в момента са реализирани като речници на Питон, но това обикновено не е забележимо по никакъв начин (с изключение на производителността), и може да се промени за в бъдеще. Примери за пространства на имената са: множеството от вградени имена (функции като `abs()`, вградените имена на изключения, глобалните имена в даден модул, и локалните имена в дадено извикване на функция. В този смисъл, множеството от атрибути на даден обект също формира пространство на имената. Важното нещо, което трябва да се знае за пространствата на имената е, че няма абсолютно никакво отношение между имената в различните пространства от имена. Например, два различни модула могат едновременно да дефинират функция "maximize", без да предизвикат объркване — потребителите на модулите трябва да сложат името на модула преди името на функцията.

Между другото, използвам думата *атрибут* за всяко име, което се употребява след точка — например, в израза `z.real`, `real` е атрибут на обекта `z`. Строго казано, обръщенията към имена в модули са обръщения към атрибути — в израза `modname.funcname`, `modname` е обект от тип модул, а `funcname` е негов атрибут. В този случай се получава пряка асоциация между атрибутите на модула и глобалните имена, дефинирани в модула: те споделят едно и също пространство![3]

Атрибутите биват само за четене или и за писане. Във вторият случай са възможни присвоявания на атрибути. В атрибутите на модулите може да се пише: например '`modname.the_answer = 42`'. Атрибутите, в които може да се пише, могат също да бъдат изтрити с оператора `del`, например: '`del modname.the_answer`'.

Пространствата на имената се създават в различни моменти и имат различен живот. Пространството на имената, съдържащо вградените имена, се създава, когато се стартира интерпретатора на Питон, и никога не се изтрива. Пространство на глобалните имена за даден модул се създава, когато се прочита дефиницията на модула; обикновено пространствата на имената за модули остават докато интерпретатора не напусне. Операторите, изпълнени в най-високото ниво на извикване на интерпретатора, независимо дали са четени от скрипт файл или интерактивно, се считат за част от модула, наречен `__main__`, така че те имат свое пространство на глобалните имена. (В действителност вградените имена също живеят в модул; той е наречен `__builtin__`.)

Пространството на локалните имена за дадена функция се създава, когато се извиква функцията, и се изтрива когато функцията се върне или предизвика изключение, което не е обработено в рамките на функцията. (В действителност, "забравяне" би било по-добър начин да се опише какво в същност се случва.) Разбира се, всяко от рекурсивните извиквания притежава свое собствено пространство на локалните имена.

Обсег е текстовият регион в една програма на Питон, където дадено пространство на имената е пряко достъпно. "Пряко достъпно" тук означава, че при неопределено обръщение към име се прави опит да се открие името в това пространство на имената.

Макар че обсегите са определени статично, те се използват динамично. По всяко време на изпълнението се използват точно три вложени обсега (тоест, точно три пространства на имената са пряко достъпни): най-вътрешният обсег, претърсван първи, съдържа локалните имена. Средният обсег, претърсван втори, съдържа глобалните имена на текущия модул. И най-външният обсег (претърсван последен) е пространството на имената, съдържащо вградените имена.

Обикновено, локалният обсег се отнася за локалните имена на (текстуално) текущата функция. Извън функциите, локалният обсег се отнася за същото пространство на имената като глобалния обсег: пространството на имената за модула. Дефинициите на класове полагат още едно пространство на имената в локалния обсег.

Важно е да се схване, че обсегите са определени текстуално: глобалният обсег на дадена функция,

[3] С изключение на едно нещо. Обектите от тип модул притежават таен атрибут само за четене, наречен `__dict__`, който връща речника, използван за да се реализира пространството на имената на модула; името `__dict__` е атрибут, а не глобално име. Очевидно, използването му нарушава абстракцията на реализацията на пространството на имената и трябва да бъде ограничено само до такива неща като дебъгери за аутопсия.

дефинирана в даден модул е пространството на имената за модула, без значение от къде или чрез какъв псевдоним е извикана функцията. От друга страна, действителното претърсване за имена се извършва динамично, по време на изпълнение — обаче, дефиницията на езика се развива по посока на статичното преобразуване на имената, по време на "компилиране", така че не разчитайте на динамичното преобразуване на имената! (На практика, локалните променливи вече се определят статично.)

Особен чалъм на Питон е, че присвояванията винаги отиват в най-вътрешния обсег. Присвояванията не копират данни — те просто свързват имена към обекти. Същото е вярно и за изтриванията: операторът '`del x`' премахва връзката на `x` от пространството на имената, към което се отнася локалния обсег. Всъщност всички операции, които въвеждат нови имена, използват локалния обсег: в частност, импортиращите оператори и дефинициите на функции свързват името на модула или функцията към локалния обсег. (Операторът `global` може да бъде използван, за да се укаже, че определени променливи пребивават в глобалния обсег.)

9.3. Пръв поглед върху класовете

Класовете въвеждат съвсем малко нов синтаксис, три нови типа обекти, и малко нова семантика.

9.3.1. Синтаксис на дефиницията на клас

Най-простата форма на дефиниция на клас изглежда така:

```
class ClassName:
    <оператор-1>
    .
    .
    .
    <оператор-N>
```

Дефинициите на класове, подобно на дефинициите на функции (оператори `def`), трябва да бъдат изпълнени преди да имат каквото и да било въздействие. (Възможно е да разположите дефиниция на клас в разклонението на даден оператор `if`, или вътре във функция.)

На практика, операторите вътре в дефиниция на клас обикновено ще бъдат дефиниции на функции, но са позволени и други оператори, което понякога е полезно, но за това — по-късно. Дефинициите на функции вътре в даден клас обикновено имат особена форма на списък на аргументи, продиктувана от конвенциите за извикване на методи — отново, това ще бъде обяснено по-късно.

Когато се въведе дефиниция на клас, се създава ново пространство на имената, и то се използва като локален обсег. Така, всички присвоявания към локални променливи отиват в това ново пространство на имената. В частност, дефинициите на функция се свързват тук с името на новата функция.

Когато дефиницията на клас се напусне нормално (през края си), тогава е създаден *клас обект (class object)*. По същество това е обвивка около съдържанието на пространството на имената, създадено от дефиницията на класа; ще научим повече за клас обектите в следващата подглава. Първоначалният локален обсег (този, който е бил в действие точно преди въвеждането на дефинициите на класа) е възстановен, и клас обектът се свързва тук с името на класа, дадено в заглавието на дефиницията на класа (`ClassName` в примера).

9.3.2. Клас обекти

Клас обектите поддържат два вида операции: обръщения към атрибути (attribute references) и създаване на инстанции (instantiation).

Обръщенията към атрибути използват стандартния синтаксис, употребяван за всички обръщения към атрибути в Питон — `обект.име`. Валидни имена на атрибути са всички онези, които са се намирали в пространството на имената на класа, когато клас обектът е бил създаден. Така че ако дефиницията на класа е изглеждала по този начин,

```
class MyClass:
    "Един прост примерен клас"
    i = 12345
    def f(x):
        return 'шкембе чорба'
```

то `MyClass.i` и `MyClass.f` са валидни обръщения към атрибути, които съответно връщат цяло число и обект от тип метод. Атрибутите на класовете също могат да бъдат присвоявани, така че можете да промените стойността на `MyClass.i` чрез присвояване. (`__doc__` също е валиден атрибут, който връща документационния символен низ, принадлежащ на класа: `"Един прост примерен клас"`).

При *създаването на инстанция* на клас се използва нотациятата на функция. Просто си представете, че клас обектът е функция без параметри, която връща нова инстанция на класа. Така например, (използвайки горния клас):

```
x = MyClass()
```

се създава нова *инстанция* на класа и присвоява този обект към локалната променлива **x**.

Операцията за създаване на инстанция ("извикването" на клас обект) създава празен обект. За предпочитание е повечето класове да създават обекти в определено начално състояние. Затова класът може да дефинира специален метод, наречен `__init__()`, например така:

```
    def __init__(self):
        self.data = []
```

Когато даден клас дефинира метод `__init__()`, тогава при създаването на инстанцията автоматично се извиква `__init__()` за новосъздадената инстанция на класа. Така в този пример, една нова, инициализирана инстанция може да бъде получена чрез:

```
x = MyClass()
```

Разбира се, за по-голяма гъвкавост методът `__init__()` може да има аргументи. В този случай аргументите, подадени на оператора за създаване на инстанция на класа, се предават на `__init__()`. Например:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0,-4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Обекти-инстанции

Какво можем да правим с обектите-инстанции (instance objects)? Единствените операции, разбираеми за тях, са обръщенията към атрибути. Съществуват два вида валидни имена на атрибути.

Първите аз наричам *атрибути-данни (data attributes)*. Те съответстват на "променливите на инстанцията" в Smalltalk, и на "членовете данни" в C++. Атрибутите-данни няма нужда да се декларират; подобно на локалните променливи, те се раждат за живот, когато за първи път им се присвоява стойност. Например, ако **x** е инстанцията на `MyClass`, създадена по-горе, следващият фрагмент код ще изведе стойността `16`, без да доведе до трасиране на стека:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Вторият вид обръщения към атрибути, разбираеми от обектите-инстанции са *методите*. Методът

е функция, която "принадлежи на" даден обект. (В Питон, терминът "метод" не е характерен единствено за инстанциите на класове — други типове обекти също могат да притежават методи. Например обектите от тип списък притежават методи, наречени append, insert, remove, sort и прочие. Все пак по-долу ще използваме термина "метод" изключително в смисъл на методи на обекти-инстанции на класове, освен ако изрично не е посочено друго.)

Валидните имена на методи за даден обект-инстанция зависят от неговия клас. По дефиниция, всички атрибути на един клас, които са (дефинирани от потребителя) обекти от тип фунцкия, дефинират съответните методи на инстанциите на класа. Така че в нашия пример, `x.f` е валидно обръщение към метод, тъй като `MyClass.f` е функция, а `x.i` не е, тъй като `MyClass.i` не е. Но `x.f` не е същото като `MyClass.f` — то е *обект от тип метод*, а не обект от тип функция.

9.3.4. Обекти от тип метод

Обикновено метод се вика непосредствено, например:

```
x.f()
```

В нашия пример той ще върне символния низ `'шкембе чорба'`. Все пак, не е задължително методът да се извиква веднага: `x.f` е обект от тип метод, и може да бъде запазен настрана, за да бъде извикан някъде по-късно. Например:

```
xf = x.f
while 1:
    print xf()
```

ще продължи да извежда '`шкембе чорба`', докато дойде края на света.

Какво точно се случва, когато е извикан метод? Може би сте забелязали, че `x.f()` беше извикан по-горе без аргумент, въпреки че дефиницията на функцията за `f` определя един аргумент. Какво се е случило с аргумента? Със сигурност Питон предизвиква изключение, когато дадена функция, изискваща аргумент, се извикава без аргументи — дори когато аргументът в действителност не се използва за нищо...

Всъщност може би сте отгатнали отговора: особеното на методите е, че обектът се предава като първи аргумент на функцията. В нашия пример извикването на `x.f()` е точен еквивалент на `MyClass.f(x)`. Изобщо, извикването на метод със списък от *n* аргумента е еквивалентно на извикване на съответната функция със списък от аргументи, в който преди първия аргумент е добавен обекта на метода.

Ако все още не разбирате как работят методите, тогава един поглед върху реализацията може би ще изясни въпроса. Когато се направи обръщение към атрибут на инстанция, който не е атрибут-данни, тогава се търси класа му. Ако името означава валиден атрибут на клас, който атрибут е обект от тип функция, тогава се създава обект от тип метод чрез пакетиране на (указатели към) обекта-инстанция и току-що намереният обект от тип функция в един абстрактен обект — това е обектът от тип метод. Когато обектът от тип метод се извика със списък от аргументи, той отново се разпакетира и се построява нов списък от аргументи с обекта инстанция и първоначалния списък от аргументи.

9.4. Нахвърляни бележки

[Може би трябва да се разположат по-внимателно...]

Атрибутите-данни припокриват атрибутите-методи със същото име; за да избегнете случайните конфликти с имената, които могат да породят трудни за откриване грешки в големи програми, разумно е да използвате някакъв вид конвенция, която намалява вероятноста от конфликти, например, започвайте с главна буква имената на методите, започвайте имената на атрибутите-данни с кратък уникален символен низ (може би долна черта), или използвайте глаголи за методите и съществителни за атрибутите-данни.

Към атрибутите-данни могат да се обръщат методи, също както и обикновени потребители ("клиенти") на обекта. С други думи, класовете не са полезни за реализиране на чисто абстрактни типове данни. Всъщност нищо в Питон не прави възможно принудителното скриване на данни — всичко е въпрос на конвенция. (От друга страна, реализацията на Питон, написана на C, може напълно да скрие подробностите на реализацията и да контролира достъпа до даден обект, ако е необходимо; това може да се използва от разширенията на Питон, написани на C.)

Клиентите би трябвало да използват атрибутите-данни внимателно — клиентите могат да объркат типовете, поддържани от методите, стъпвайки върху техните атрибути-данни. Забележете, че клиентите могат "на своя глава" да добавят атрибути-данни към обекти-инстанции, без да засегнат валидността на методите, стига да се избягват конфликтите с имена — и тук една конвенция за именуване тук ще спести много главоболия.

В рамките на методите не съществува кратък начин за обръщение към атрибутите-данни (или други методи!). Намирам, че това действително увеличава четимостта на методите: няма вероятност да объркате локалните променливи и променливите на инстанцията, когато хвърлите поглед на даден метод.

Общоприето е първият аргумент на методите да се нарича `self`. Това не е нищо повече от една конвенция: името `self` няма абсолютно никакво особено значение за Питон. (Забележете, обаче, че ако не следвате конвенцията, кодът Ви може да стане по-нечетим за други програмисти на Питон, а също така е възможно някоя програма за *преглед на класовете* (*class browser*) да бъде написана така, че да разчита на тази конвенция.)

Всеки обект от тип функция, който е атрибут на клас, дефинира метод за инстанциите на този клас. Не е задължително дефиницията на функцията да е заградена текстуално в дефиницията на класа: присвояването на обекта от тип функция към локална променлива в класа също работи. Например:

```
# Дефиниция на функция извън класа
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'шкембе чорба'
    h = g
```

Сега `f`, `g` и `h` са атрибути на класа `C`, които се отнасят до обекти от тип функция, и следователно всички те са методи на инстанциите на `C` — при това `h` е точен еквивалент на `g`. Забележете, че тази практика обикновено служи само за объркване на този, който чете програмата.

Методите могат да извикват други методи, използвайки метод-атрибутите на аргумента `self`, например:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Методите могат да се обръщат към глобални имена по същия начин, както обикновените функции. Асоцииранията с метода глобален обсег е модулът, съдържащ дефиницията на класа. (Самият клас никога не се използва като глобален обсег!) Докато някои рядко срещат основателна причина за използване на глобални данни в метод, съществуват много легитимни употреби на глобалния обсег: най-вече, импортираните в глобалния обсег функции и модули могат да бъдат използвани от методите, тъй както дефинираните в него функции и класове. Обикновено самият клас, съдържащ метода, е дефиниран в този глобален обсег, и в следващата подглава ще намерим няколко основателни причини за това, че един метод би желал да се обръща към собствения си клас!

9.5. Наследяване

Разбира се, една възможност на езика няма да е достойна за името "клас" без да поддържа наследяване. Синтаксисът за дефиниция на произхождащ клас изглежда така:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Името `BaseClassName` трябва да е дефинирано в обсега, съдържащ дефиницията на произхождащия клас. Допустим е и израз вместо име на базов клас. Това е полезно, когато базовият клас е дефиниран в друг модул, например:

```
class DerivedClassName(modname.BaseClassName):
```

Изпълнението на дефиницията на произхождащ клас преминава по същия начин, както за базов клас. Когато клас обектът е създаден, базовият клас се запомня. Той се използва за преобразуване на обръщения към атрибути: ако изисканият атрибут не е открит в класа, за него се претърсва базовият клас. Правилото се прилага рекурсивно ако базовият клас сам произхожда от някакъв друг клас.

Няма нищо особено относно създаването на инстанция на произхождащи класове: `DerivedClassName()` създава нова инстанция на класа. Обръщенията към методи се преобразуават както следва: претърсва се за съответния атрибут на клас, ако е необходимо слизайки надолу по веригата на базовите класове, и обръщението към метода е валидно, ако то води до обект от тип функция.

Произхождащите класове могат да припокриват методи на базовите си класове. Понеже методите нямат особени привилегии когато се извикват други методи на същия обект, даден метод на базов клас, който вика друг метод, дефиниран в същия базов клас, може в действителност да свърши с извикване на припокрилия го метод от произхождащия клас. (За програмистите на C++: всички методи в Питон действително са `virtual`).

Припокриването на метод в произхождащия клас може всъщност да цели разширяване, вместо просто заменяне на метода на базовия клас с друг със същото име. Съществува прост начин за извикване на базовия клас направо: просто извикайте '`BaseClassName.methodname(self, arguments)`'. Това понякога е полезно и за клиентите. (Забележете, че това работи само ако името на базовия кла е дефинирано или импортирано направо в глобалния обсег.)

9.5.1. Множествено наследяване

Питон поддържа и една ограничена форма на множествено наследяване. Дефиницията на клас с няколко базови класове изглежда както следва:

```
class DerivedClassName(Base1, Base2, Base3):
    <оператор-1>
    .
    .
    .
    <оператор-N>
```

За да се обясни семантиката, единственото необходимо правило е това, използвано за взимане на решение при обръщения към атрибути на клас. То е първо в дълбочина, сетне от ляво на дясно. По такъв начин, ако даден атрибут не е намерен в `DerivedClassName`, за него се претърсва в `Base1`, после (рекурсивно) в базовите класове на `Base1`, и само ако атрибутът не е открит там, тогава се претърсва `Base2`, и така нататък.

(За някои хора, претърсването първо по широчина, т.е. претърсването на `Base2` и `Base3` преди базовите класове на `Base1`, изглежда по-естествено. Обаче, това би изисквало от Вас да знаете дали определен атрибут на `Base1` е действително дефиниран в `Base1` или в някой от базовите му класове, за да пресметнете последствията от конфликт с името на атрибут от `Base2`. Правилото за претърсване първо по дълбочина не прави разлика между преки и наследени атрибути на `Base1`.)

Ясно е, че една объркана употреба на множественото наследяване ще е кошмарна за поддръжка, дължащо се на това, че в Питон се разчита на конвенциите за да се избягват евентуални конфликти с имената. Добре познат проблем с множественото наследяване е когато клас произхожда от два класа,

които се случва да имат общ базов клас. Докато е достатъчно лесно да се пресметне какво става в този случай (инстанцията ще има едно копие на "променливите на инстанцията" или атрибутите-данни използвани от общия базов клас), то не е ясно по какъв начин изобщо е полезна такава семантика.

9.6. Частни променливи

Съществува ограничена поддръжка на частни (private) идентификатори за клас. Всеки идентификатор във формата `__spam` (поне две долни черти в началото, най-много една долна черта накрая) сега текстуално се заменя с `_classname__spam`, където `classname` е името на текущия клас с премахнати начални долни черти. Това кълцане се извършва без оглед на синтактичната позиция на идентификатора, така че може да бъде използвана за да се дефинират частни за класа инстанции и променливи на класа, методи, както и глобални променливи, и дори да се съхраняват променливи с инстанции, частни за този клас, инстанции на *други* класове.

Кълцането на имената е предназначено да предостави лесен начин за дефиниране на "частни" променливи на инстанции и методи, без да е необходимо да се тревожим за променливите на инстанциите, дефинирани в произхождащите класове, или за оплескване на променливите на инстанциите от код извън класа. Забележете, че правилата за кълцане са проектирани най-вече за да се избегнат случайности; за решителната душа все пак е възможно да получи достъп или да промени променлива, считана за частна. Това дори може да бъде полезно, например за дебъгер, и това е една от причините, поради която тази вратичка не е затворена. (Скрита опастност за грешка: произхождащ клас със същото име, както името на базовия клас, може да направи възможно използването на частните променливи на базовия клас.)

Обърнете внимание, че кодът, подаден на `exec`, `eval()` или `evalfile()`, не счита името на извикващия клас за текущ клас; това е ефект, подобен на този на оператора `global`, резултатът от който също е ограничен до кода с който е байт-компилиран заедно. Същото ограничение се отнася за `getattr()`, `setattr()` и `delattr()`, тъй както и когато се прави обръщение директно към `__dict__`.

Ето един пример за клас, който реализира свои собствени методи `__getattr__` и `__setattr__` и съхранява всички атрибути в частна променлива по начин, който работи в Питон 1.4 тъй както и в предните версии:

```
class VirtualAttributes:
    __vdict = None
    __vdict_name = locals().keys()[0]

    def __init__(self):
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

9.7. Туй-онуй

Понякога е полезно да разполагате с тип данни, подобен на "record" в Паскал или "struct" в C, за да пакетирате заедно няколко именовани елемента данни. Една празна дефиниция на клас ще свърши чудесна работа, например:

```
class Rabotnik:
    pass

elijah = Rabotnik() # Създава празен запис за работник

# Fill the fields of the record
elijah.ime = 'Илия Стойков'
elijah.otdel = 'софтуер'
elijah.zaplata = 1000
```

Ако част от код на Питон очаква определен абстрактен тип данни, често може да бъде предаден клас, който вместо това емулира методите на този тип данни. Например, ако разполагате с функция, която форматира някакви данни от файлов обект, можете да дефинирате клас с методи `read()` и `readline()`, които вместо да четат от файл, взимат данните от буфер със символни низове, и да подадете този клас като аргумент.

Обектите от тип метод също имат атрибути: `m.im_self` е обектът, към който принадлежи този метод, а `m.im_func` е обектът от тип функция, който съответства на метода.

9.7.1. Изключенията могат да бъдат класове

Дефинираните от потребителя изключения вече не са ограничени до това да бъдат обекти от тип символен низ — те могат да бъдат идентифицирани и като класове. Използвайки този механизъм е възможно да се създаде разширяема йерархия от изключения:

Съществуват две нови валидни (семантични) форми на оператора raise:

```
raise Class, instance

raise instance
```

В първата форма, `instance` трябва да бъде инстанция на `Class` или на някой клас, произхождащ от него. Втората форма е кратък запис на:

```
raise instance.__class__, instance
```

Една except клауза може да изброява класове, тъй както и обекти от тип символен низ. Даден клас в except клауза е съвместим с дадено изключение, ако то е от същия клас или вместо това е наследник на класа (но не и обратното — except клауза, изброяваща произхождащ клас, не е съвместима с базовия клас). Например, следващият код ще изведе B, C, D в този ред:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Забележете, че ако except клаузите бяха в обратен ред (с '`except B`' в началото), би се извело B, B, B — стартира се първата съвпадаща except клауза.

Когато се изведе съобщение за грешка за необработено изключение, което е клас, извежда се името на класа, после двоеточие и интервал, и накрая инстанцията, превърната в символен низ чрез вградената функция `str()`.

ГЛАВА
ДЕСЕТА

Сега какво?

Надявам се, че прочитането на това ръководство е подсилило интереса Ви да използвате Питон. Какво би трябвало да направите оттук нататък?

Би трябвало да прочетете или поне да прелистите Library Reference, който дава пълен (макар и сбит) справочен материал за типове, функции, и модули, и може да Ви спести много време докато пишете програми на Питон. Стандартната дистрибуция на Питон включва *много* код на C и Питон. Там има модули за четене на Unix пощенски кутии, извличане на документи чрез HTTP, генериране на случайни числа, граматичен разбор на опции от командния ред, писане на CGI програми, компресиране на данни, и много други. Прелистването на Library Reference ще ви даде представа с какво разполагате.

Главното web-място на Питон е http://www.python.org; то съдържа код, документация, и указатели към свързани с Питон страници из Мрежата. Това web-място има огледални копия (mirror) в различни места на света, като Европа, Япония, и Австралия; огледалното копие може да бъде по-бързо от основното място, в зависимост от географското Ви местоположение. По-неофициално място е http://starship.python.net/, съдържащо цял куп от лични страници, свързани с Питон; тук много хора разполагат софтуер за сваляне.

Свързаните с Питон въпроси и доклади за проблеми можете да пускате в новинарската група (newsgroup) comp.lang.python, или да изпращате в пощенския списък (mailing list) на python-list@cwi.nl. Между новинарската група и пощенския списък има шлюз, така че изпратените съобщения до едното, автоматично ще се препратят до другото. Там има около 35–45 нови съобщения дневно, които задават (или отговарят на) въпроси, предлагат нови възможности, анонсират нови модули. Преди да пуснете съобщение, уверете се че сте проверили списъка на Често Задаваните Въпроси (Frequently Asked Questions, наричан още FAQ), или погледнете в директорията 'Misc/' на дистрибуцията на Питон с изходен код. FAQ отговаря на много въпроси, които се задават отново и отново, и може би вече съдържа решение на Вашия проблем.

Можете да поддържате обществото на Питон като се присъедините към Python Software Activity, която обслужва web, ftp и пощенските услуги на python.org, и организира работилниците на Питон. Вижте http://www.python.org/psa/ за информация за това как да се присъедините.

ПРИЛОЖЕНИЕ
A

Интерактивно редактиране на входа и заместване с история

Някои версии на интерпретатора на Питон поддържат редактиране на текущия входен ред и заместване с история (history substitution), подобно на средствата, които откриваме в обвивката на Корн и обвивката GNU Bash. Това е реализирано чрез библиотеката *GNU Readline*, която поддържа редактиране по маниера на Emacs и по маниера на vi. Тази библиотека има своя собствена документация, която няма да повтарям тук; обаче основните неща са обяснени накратко. Интерактивното редактиране и история, описани тук, са достъпни като опция във версиите на интерпретатора за Unix и CygWin.

Тази глава *не* документира редактиращите средства на пакета PythonWin на Mark Hammond или на Tk-базираната среда IDLE, разпространявана с Питон. Връщането по историята на командния ред, работещо в DOS прозорци върху NT и някои други DOS и Windows варианти, също се различава.

A.1. Редактиране на реда

Ако се поддържа, редактирането на входния ред е активно всеки път, когато интерпретаторът изведе първичен или вторичен промпт. Текущият ред може да бъде редактиран с обичайните контролни знаци на Emacs. Най-важните от тях са: C-A (Control-A) премества курсора в началото на реда, C-E в края[1], C-B го премества една позиция в ляво[2], C-F в дясно[3]. Backspace изтрива знака в ляво от курсора, C-D знака от дясно. C-K убива[4] (изтрива) остатъка от реда в дясно от курсора, C-Y дърпа обратно[5] последния убит символен низ. C-долна_черта връща обратно последната извършена промяна; това може да бъде повтаряно за да се получи натрупване на ефекта.

A.2. Заместване с история

Заместването с история работи така. Всички подадени непразни входни редове се запазват в буфер на историята, и когато ви се подаде промпт, Вие сте поставени на нов ред в края на този буфер. C-P премества с един ред нагоре (назад)[6] в буфера на историята, C-N премества с един надолу[7]. Всеки ред в буфера на историята може да бъде редактиран; една звездичка се появява в началото на промпта за да обозначи, че реда е модифициран. Натискането на клавиша Return предава текущия ред на интерпретатора. C-R започва обратно нарастващо търсене[8]; C-S започва търсене напред[9].

A.3. Значение на клавишните комбинации

Значението на клавишните комбинации и някои други параметри на билбиотеката Readline могат да бъдат настройвани чрез разполагането на команди в един инициализационен файл, наречен

[1] от англ. end — край *(бел. прев.)*
[2] от англ. back — назад *(бел. прев.)*
[3] от англ. forward — напред *(бел. прев.)*
[4] от англ. kill — убивам *(бел. прев.)*
[5] от англ. yank — дърпам, издърпвам *(бел. прев.)*
[6] от англ. previous — предишен *(бел. прев.)*
[7] от англ. next — следващ *(бел. прев.)*
[8] от англ. reverse — обратно *(бел. прев.)*
[9] от англ. search — търся *(бел. прев.)*

‘$HOME/.inputrc’. Значенията на клавишните комбинации имат формата:

```
име-на-клавишна-комбинация: име-на-функция
```

или

```
"символен низ": име-на-функция
```

и опциите могат да бъдат установявани чрез

```
set име-на-опция стойност
```

Например:

```
# Аз предпочитам редактиране по маниера на vi:
set editing-mode vi
# Редактиране на един ред:
set horizontal-scroll-mode On
# Да предефинираме няколко клавишни комбинации:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Забележете, че по подразбиране значението на TAB в Питон е вмъкване на TAB, вместо традиционната за Readline функция за завършване на файлово име. Ако настоявате, можете да припокриете това чрез поставянето на

```
TAB: complete
```

във Вашия ‘$HOME/.inputrc’. (Разбира се, това затруднява въвеждането на продължаващи редове с отстъп...)

Като опция се предлага автоматично завършване на имената на променливи и модули. За да го включите в интерактивния режим на интерпретатора, във Вашия ‘$HOME/.pythonrc.py’ добавете следното:

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Това прикрепя клавиша TAB към функцията за завършване, така че удрянето на клавиша TAB два пъти предлага завършени имена; функцията преглежда имената на оператори на Питон, текущите локални променливи, и достъпните имена на модули. За “точкувани” изрази като `string.a`, ще се изчисли израза до последната ‘`.`’ и после ще бъдат предложени завършени имена из атрибутите на получения обект. Забележете, че така може да се изпълни дефиниран от приложението код, ако част от израза е обект, дефинирал метод `__getattr__()`.

A.4. Коментар

Това средство е огромна стъпка напред в сравнение с предишните версии на интерпретатора; останали са, обаче, някои желания: Би било чудесно ако правилният отстъп бе предлаган от продължителните редове (граматичният анализатор знае дали следва да се изисква знак за отстъп). Механизмът за завършване може да използва символната таблица на интерпретатора. Също така би била полезна команда за проверяване (или дори предлагане) на съответстващи скоби, кавички, и прочие.