

Введение

Основная цель данной книги состоит в том, чтобы научить Вас программированию. И только во вторую очередь преследуется цель преподать Вам основы Python, вероятно, одного из наиболее стильных языков программирования на планете. Вам совершенно не обязательно что-либо знать о программировании до начала обучения. Все, что Вам необходимо, — это желание учиться. Если у Вас имеется компьютер и выход в Internet, Вы можете получить Python бесплатно. В главе 1 подробно описано, как получить и установить Python на своём компьютере. Python выполняется на различных операционных системах, причём его переносимость является рекордной по сравнению с остальными языками программирования. Так что Вы обязательно найдёте версию, которая подойдёт для вашей системы, будь то Windows, UNIX, Amiga или Macintosh.

***Прим. В. Шипкова: Более того, в Линуксе этот язык программирования идёт в комплекте, мне даже доводилось слышать о реализации под QNX — ОС реального времени, родственница UNIX. Вообще, складывается впечатление, что Оси делятся на три группы: UNIX и *nix'ы, Windows9x и Windows NT, [MenuetOS, AsmOS, TriadaOS... и другие микро-оси].**

Если прежде Вы никогда не программировали, то имеете значительное преимущество. Многие люди приступают к работе с Python только после того, как потратили несколько лет, пытаясь проникнуть в тайны и лабиринты других, более сложных языков программирования. Им приходится начинать с того, чтобы постараться забыть как можно больше обо всех остальных языках. Вам же не придётся преодолевать собственное предвзятое мнение, ошибочные концепции и привычки. Вы имеете то, что в философии Дзен называется "мышлением новичка". Преднамеренное введение себя в такое состояние считается первым шагом к овладению Дзен. Каждый раз, когда мастера Дзен приступают к практическим занятиям или медитации, они стремятся удержаться от того, чтобы уже имеющиеся знания повлияли каким-то образом на их сознание. Ничто так не мешает постижению новых знаний, как груз сложившихся предвзятых отношений и оценок. Часто, чтобы увидеть новое, достаточно посмотреть на старое с новой точки зрения. Писателям также известен этот принцип. Тот

факт, что однажды Вы уже написали книгу, совсем не означает, что Вы знаете, как писать следующую. Также и каждый язык программирования требует свежих мыслей и подходов. Но дело в том, что овладение программированием станет для Вас проще, концепции понятнее, а переход к другим языкам менее болезненным, если первым вашим языком программирования станет Python. Благодаря простоте, четкости, ясности и эффективности Python признан языком, способствующим привитию хороших навыков объектно-ориентированного программирования.

Принципы программирования на Python предельно просты для понимания, но сосредоточьтесь на практической работе с Python. Постарайтесь выполнять все задания, предлагаемые в конце каждой главы. Не поддавайтесь соблазну забежать вперед до того, как Вы уверенно усвоите и закрепите на практике материал текущей главы. Это может показаться слишком тривиальным советом, настолько очевидным, что граничит с занудством автора. Но Вы не представляете, сколько людей пытались программировать, но у них ничего не получилось. А все потому, что их мысли оказались занятыми не работой над языком программирования, а рассуждениями о том, сколько денег они начнут зарабатывать и какие огромные проблемы решат, стоит только им все бросить и прямо сейчас взяться за сложные проекты.

Умоляю Вас, на каждом занятии думайте только о занятии, а головокружительные проекты оставьте на потом, иначе Вас ждет разочарование. Если же Вы будете внимательно относиться к урокам и практическим занятиям, то сможете стать хорошим программистом, но не более того. Чтобы стать великим программистом, мало хорошо знать средства того или иного языка, важно уметь думать на нем, ощущать его концепции и уметь выражать свои мысли с помощью этого языка программирования.

Python – идеальный язык для обучения программированию. Его синтаксис прост, понятен и в то же время обладает огромной мощностью. Программный код логичен и ясен. Он полностью лишен того, что на сленге программистов называется "шаманизмом", т.е. программных решений, которые невозможно понять с точки зрения здравого смысла, а можно только запомнить. Именно поэтому говорят, что мышление Python – это мышление новичка в программировании. Python не может устранять за Вас ваши ошибки, но благодаря ясности и логичности кода он затруднит их появление, облегчит чтение и анализ программ, разработанных другими программистами.

Как пользоваться этой книгой

Книга содержит 24 главы, которые объединены в три части. Для полного усвоения материала каждого занятия потребуется

примерно один час времени. Если у Вас уходит немного больше или меньше времени, значит, это именно то время, которое требуется лично Вам. В выделении трёх частей книги я следовал тактике автора известной книги о Дзен-буддизме Шанрай Сузуки (Shunryu Suzuki) *Zen Mind, Beginner's Mind* ("Мышление Дзен — мышление новичка"). (Впрочем, чтобы научиться программированию на Python, Вам совсем не обязательно начинать с этой книги. Это лишь мой собственный путь познания.) Книга Сузуки также разбита на три части, следующие в таком порядке: *Right Practice* (Правильная практика), *Right Attitude* (Правильное отношение) и *Right Understanding* (Верное понимание). Три части данной книги познакомят Вас (строго по порядку) с основными средствами программирования Python, концепциями объектно-ориентированного программирования и использованием всех этих знаний в Python на примере построения графического интерфейса пользователя. Если во всем этом Вы не найдёте аналогии с Дзен-буддизмом, значит, у нас просто разное мировоззрение.

Вопросы и ответы, контрольные вопросы, примеры и задания

Каждая глава книги заканчивается тремя следующими разделами.

- Вопросы и ответы — несколько часто задаваемых вопросов и ответы на них по материалу, пройденному в этой главе.
- Контрольные вопросы — несколько вопросов для проверки ваших знаний.
- Примеры и задания — дополнительные и необязательные (но весьма полезные) примеры и задания для закрепления практических навыков в Python.

Эти разделы необязательны для чтения и выполнения, но весьма полезны, поэтому старайтесь не пропускать их.

Первоисточники Python в Internet

Python — это язык, который изменяется медленно, но постоянно, как и было задумано его создателем Гуидо ван Россумом (Guido van Rossum), который всеми силами пытается сочетать прогрессивность новых идей с преемственностью версий. Изменения вносятся в язык только после многочисленных обсуждений и дебатов, но при этом всегда соблюдается одно правило — программы, написанные раньше, должны продолжать работать с новой версией Python. Чтобы быть в курсе самых последних новостей о Python, посещайте официальную домашнюю страницу Python по адресу <http://www.python.org/>. Имеется ещё один чрезвычайно

полезный ресурс о Python – сервер *The Vaults of Parnassus* (Пещеры Парнаса) по адресу <http://www.vex.net/parnassus/>.

Часть 1

Python мыслит так, как мыслит новичок

Темы занятий

1. Общие представления о языке Python
2. Интерпретатор Python
3. Арифметические действия в Python
4. Переменные и управление ими
5. Основные типы данных: числовые
6. Основные типы данных: последовательности и словари
7. Функции и модули
8. Полезные средства и приёмы программирования

1-й час

Общие представления о языке Python

В данной главе Вы получите общее представление о Python: что это за язык, какова его история, в каких случаях он хорош и где неприменим. Для усвоения материала этой части Вам даже не понадобится компьютер, пока Вы не дойдёте до раздела "Примеры и задания", где предложен список полезных Web-страниц.

Зачем вообще программировать и почему на Python

Если требования, предъявляемые к возможностям Вашего компьютера, ограничиваются всего лишь сведением баланса чековой книжки, Вам совершенно не обязательно знать программирование. Для этого имеются более приемлемые инструментальные средства, например карандаш, бумага и калькулятор. А если Вы используете компьютер только для редактирования и верстки текстов, то опять же, нет никакой нужды изучать программирование. Имеется множество программ, которые превосходно справятся с любой вашей прихотью и выполнят все, что Вы захотите.

Но если вдруг окажется, что для осуществления ваших замыслов не найдется соответствующего программного обеспечения, или то, что имеется, не удовлетворяет вашим запросам, единственный выход – написать своё собственное приложение. Вероятно, эта простая идея и привела человечество к новым и значительным прорывам в



программировании и появлению более качественных программ. Отличный пример тому – операционная система Linux. Линус Торвалдс (Linus Torvalds), удрученный существующими реализациями UNIX для компьютеров на платформе PC, решился написать свою собственную версию. Сегодня Linux стала настолько популярной, что на неё с беспокойством оглядывается даже Билл Гейтс со своей компанией Microsoft.

***Прим. В. Шипкова: Это не преувеличение. На конец 2005 года популярность платформы на столько возросла, что такие монстры, как IBM, Oracle, Sun Microsystems вкладывают в платформу в общей сложности от 2 до 4 млрд. долларов.**

UNIX была разработана в начале семидесятых в лаборатории AT&T в Мюррей Хил (Murray Hill), штат Нью-Джерси. Мощная многопользовательская операционная система UNIX – это детище Денниса Ритчи (Dennis Ritchie), Брайана Кернигана (Brian Kernighan) и Кена Томпсона (Ken Thompson), у которых была масса свободного времени и лишний компьютер, на котором никто, кроме них, не хотел работать. До конца восьмидесятых даже такой, по нашим меркам, маломощный компьютер стоил слишком дорого, чтобы появиться в домах людей, увлеченных информатикой. Те, кто повседневно работал с UNIX, с улыбкой сожаления смотрел на слабенькие операционные системы, применяемые в персональных компьютерах. Мы были разбалованы, где-то даже испорчены нашей "большой железкой". Однако в 1987 г. Эндрю Танненбаум (Andrew Tannenbaum) разработал очень маленькую UNIX-подобную операционную систему, которая могла работать на домашних PC. Он назвал её Minix. Позже Линус Торвалдс разработал переносимую и ещё более мощную версию операционной системы, которую назвал Linux. По мощности Linux не уступала производственным операционным системам, но работала даже на дешевых домашних персональных компьютерах. Основное отличие между Minix и Linux (особенно в первое время) состояло в том, что условия лицензирования Minix содержали больше ограничений, чем при лицензировании Linux. Сегодня отличие состоит в том, что во всем мире имеются тысячи поклонников Linux, реализовавших для своей любимой системы практически все приложения, известные в мире.

Научные работники часто ощущают потребность в программах, которые в этот момент отсутствуют, и для продвижения своих исследовательских проектов нередко пишут собственные приложения. Хотя я не отношу себя к ученым, но не могу себе отказать в проведении собственных исследований. У меня сложилось твердое убеждение, что календарь индейцев Майя – это нечто восхитительное и захватывающее. В течение

нескольких лет я создавал программы на языке C, которые помогли мне исследовать эту столь интересную тему. Когда я обнаружил Python, то быстро отказался от всей массы приложений, которую успел наваять, и повторно реализовал все программы с помощью языка Python. Завершив свою работу, я обнаружил, что все модули и библиотеки читаются проще и яснее, уменьшились в размере и стали значительно более мощными. Более того, все, что мне было необходимо, я смог создать значительно быстрее, чем когда упражнялся на C.

Многие люди часто приходят к выводу, что им необходимо изучить как минимум основы программирования, чтобы автоматизировать некоторые регулярно повторяющиеся рутинные задачи. В качестве примера можно привести небольшие программки, которые собирают еженедельные отчёты сотрудников коллектива из специального каталога или папки, затем проверяют время обновления отчёта, после чего осуществляют какую-нибудь простейшую обработку. Например, объединяют индивидуальные отчёты в один сводный и распечатывают полученный результат или отсылают его по электронной почте руководителю группы. Мне приходилось заниматься подобными задачами в самом начале своей карьеры программиста. Я справлялся со своими обязанностями, но приходилось повозиться, даже использовать несколько различных языков написания сценариев. Если бы в то время существовал Python, то я смог бы выполнить все задачи значительно быстрее и существенно уменьшить число строк в коде, причём опираясь только на этот язык программирования.

Приведу небольшой перечень некоторых регулярно повторяющихся рутинных задач, которые мне приходилось решать на протяжении многих лет своей работы. Практически все они могут быть вполне удовлетворительно реализованы с помощью Python:

- собирать отчеты и обобщать их в сводный отчет;
- проверять действенность гиперссылок в документах для Web;
- периодически выполнять резервное копирование файлов и каталогов;
- автоматически посылать по электронной почте отчёт своему начальнику, чтобы он пребывал в умиротворенном состоянии, полагая, что Вы действительно делаете что-то важное;
- на основе простых входных данных автоматически создавать графики в PERT (Project Evaluation and Review Technique — планирование с использованием сетевого графика);

- составлять список всех файлов в указанном дереве каталогов и выполнять различные операции с каждым файлом в зависимости от его расширения;
- создавать списки файлов, отсортированных по имени или типу;
- вести учет своей коллекции видеофильмов.

В былые времена, чтобы написать свою собственную программу для решения какой-либо задачи, нужно было изучить массу литературы о сложном и непонятном синтаксисе языка. Возьмём для примера FORTRAN – один из самых старых, но до сих пор популярных языков программирования. Он хорошо оснащен средствами для решения многих научных задач и сложных математических вычислений, но его синтаксис, мягко говоря, не совсем очевиден. Другой язык программирования – С. Вы о нем, безусловно, уже слышали. Язык С может гордиться числом приверженцев во всем мире, сердца которых он завоевал мощью и эффективностью средств программирования. Но с этим языком не успеешь оглянуться, как программа становится мудрёной, запутанной и трудно читаемой. Профессиональные программисты признают, что язык С прививает не лучшие навыки программирования, а FORTRAN навязывает свои правила игры. В противоположность им Python стимулирует своих поклонников к приобретению хороших навыков программирования, но никого ни к чему не принуждает. ещё одна отличительная черта Python состоит в том, что ему легко учиться, поскольку стиль этого языка в наибольшей мере соответствует представлениям новичков о программировании.

Давайте рассмотрим несколько выполняемых программ, написанных на FORTRAN, С и Python. Конечно, они не обладают большой функциональностью. Традиционная программа для начинающих на любом языке программирования просто выводит на экран фразу "Hello, World" (Привет, мир), и результат её работы можно увидеть своими глазами на экране монитора.

***Прим. В. Шипкова: здесь и далее я тексты программ выделял в отдельные блоки. Как я уже в начале высказался – в первоисточнике неправильно было сделано форматирование, непропечатаны многие символы, что однозначно, приводило бы к невозможности выполнять программы.**

FORTRAN:

```
PROGRAM
PRINT *, "Hello, World"
END PROGRAM
```


C:

```
#include <stdio.h>
main()
{
    printf("Hello, World\n");
}
```

Python:

```
print "Hello, World"
```

Я считаю, что Python ближе всех подошел к тому, что можно назвать идеальным языком программирования. Его синтаксис предельно прост и понятен. Вместо концепции множества решений одной задачи в нем, как правило, предлагается одно или несколько очевидных и наилучших решений. Причем многие особо полезные методы, необходимые для научного программирования, уже встроены в интерпретатор Python и устанавливаются вместе с ним. Это чрезвычайно удобно, так как не нужно тратить время на поиск модулей независимых изготовителей и на их установку. Но для выполнения более сложных задач программирования Вам действительно понадобится дополнительный пакет под названием Tcl/Tk. Пользователям Windows повезло, так как этот пакет входит в состав дистрибутива и устанавливается автоматически при установке Python. Несмотря на кажущуюся простоту языка, Python позволяет моделировать сложные процессы и взаимоотношения благодаря тому, что в нем реализованы концепции объектно-ориентированного программирования (ООП).

В следующих главах я постараюсь донести до Вас основы программирования на Python. Первая треть книги охватывает большую часть базовых элементов Python. Следующая треть полностью посвящена объектам, которые играют важнейшую роль в реализации всей мощи данного языка. Последняя треть книги посвящена использованию переносимого графического интерфейса пользователя — *tkinter*. В последней трети мы также немного коснемся вопросов программирования с применением интерфейса CGI (common gateway interface — интерфейс общего шлюза), позволяющего программировать для Internet. Это убедит Вас в перспективности программирования на Python и необходимости продолжить обучение.

При изложении материала в этой книге я исхожу из того, что Вы ничего не знаете о программировании. Отсутствие предвзятого мнения и излишних знаний, почерпнутых из других языков программирования, окажется для Вас даже полезным. Я

надеюсь, что для Вас быстро станет очевидным тот факт, что программирование значительно проще, чем это принято считать. Тем не менее я все же полагаю, что Вы умеете пользоваться компьютером и имеете базовые знания хотя бы в таких программах, как текстовые редакторы, и об интерфейсе командной строки вашей операционной оболочки (DOS или другой, такой как ksh, csh или bash для UNIX).

Хотя уже много сказано об объективных причинах, побуждающих людей к программированию, последний штрих, который я хочу добавить, состоит в том, что создание работоспособных программ — это ещё и увлекательно. Ничто не может сравниться с тем чувством удовлетворения, которое охватывает Вас от ощущения способности расставить в определенном порядке инструкции и наблюдать, что их работа точно соответствует тому, что Вы задумали. Если раньше компьютер диктовал Вам свои условия, когда Вы обучались работе с тем или иным приложением, то теперь Вы указываете компьютеру, как он должен себя вести. Программирование на Python никогда не было для меня только работой. Надеюсь, что когда Вы закончите чтение данной книги, то сможете получать от программирования столько же удовольствия, сколько получаю я.

История Python

Python был разработан в конце 1989 г. Гуидо ван Россумом (Guido van Rossum) во время рождественских каникул, когда его исследовательская лаборатория была закрыта и ему просто некуда было деваться. Он позаимствовал многие средства программирования, присущие другим языкам, таким как ABC, Modula-3, C и некоторым другим (по крайней мере, наиболее общие средства). Он обожал телевизионную передачу *Monty Python's Flying Circus* (Летающий цирк питона Монти), и когда пришло время дать название своему языку, он выбрал имя Python. В 1991 г. после испытаний и экспериментов в узком кругу друзей и коллег по работе Python был размещен в домене общего доступа на суд широкой общественности. В отличие от других языков программирования, Python не только распространяется совершенно бесплатно, он не имеет абсолютно никаких ограничений в условиях применения. Никто не ограничивает коммерческое использование программных продуктов, написанных на этом языке, без каких-либо лицензионных отчислений. Программисты также вольны модернизировать язык, не ставя в известность автора.

В дополнение к перечисленным выше преимуществам синтаксиса само по себе размещение интерпретатора Python в домене общего доступа способствовало быстрому

распространению и росту популярности этого языка во всем мире. Другие языки, возможно, имеют большую армию пользователей, но немногие языки могут похвастать таким же широким сообществом пылких почитателей, какое имеет Python. И хотя Python ещё молодой язык программирования, его пользователи регулярно собираются на международные конференции (как минимум один раз в год, а иногда и чаще).

***Прим. В. Шипкова: совсем недавно такая конференция проходило в Москве. Очень жаль, но я туда не попал.**

Пользователи Python создали неформальную организацию, призванную поддерживать и повсеместно расширять применение приложений на Python. Эта организация называется PSA (Python Software Activity). Она объединяет более 300 человек и 30 предприятий и существует в основном на пожертвования частных лиц и предприятий. Членство не является обязательным, лишь бы Вы любили Python. Никто и никогда не обязан платить ни копейки за использование Python в каких бы то ни было целях. Поэтому просто замечательно, что были сделаны столь значительные денежные вклады для развития и роста этого языка программирования.

Первое время очень часто задавали такой вопрос: "Что произойдет, если Guido однажды потеряет ко всему этому интерес?" Поклонники Python были обеспокоены тем, что если однажды с Guido что-нибудь случится, то язык умрет. Поэтому в 1998 г. был основан консорциум, отвечающий за поддержку и развитие языка Python. Корпоративные члены консорциума выделили средства для Guido, чтобы он продолжал работу над Python (и больше ни над чем другим), а также нашел преемника к тому времени, когда у него пропадет желание или иссякнут силы управлять развитием Python.

Guido характеризует себя как "консервативного программиста", уверенного в том, что Python будет изменяться под его руководством и только в тех направлениях, которые, по его мнению, являются необходимыми. Основное достоинство такой перспективы состоит в том, что программы, написанные в ранних версиях Python, будут по-прежнему выполняться в неизменном виде и в последующих версиях (по крайней мере, большая часть кода). Лично я начал пользоваться Python с версии 1.3, а сейчас уже вышла в свет версия 2.0,

***Прим. В. Шипкова: на ноябрь-2005 - последняя версия Python 2.4.1**



но все программы, написанные мной ранее, успешно выполняются во всех версиях. Версия 1.5.2 была взята за основу при создании рисунков для этой книги, но Вы можете все также бесплатно выгрузить и установить на своём компьютере последнюю версию. Гарантирую, что все коды, приведенные в этой книге, будут выполняться без внесения каких-либо изменений. Большинство кодов также прошли проверку на более ранних версиях Python.

Инструкция goto как враг программирования

В 1968 г. Эдсгер У. Дейкстра (Edsger W. Dijkstra), один из величайших столпов программирования, написал письмо редактору издания *Communications of the ACM*, в котором чётко обосновывал, что инструкция goto, присутствовавшая в то время практически во всех языках программирования, оказывает неблагоприятный эффект на ход мышления программистов .

Инструкция компьютерного языка goto сообщает компьютеру, что необходимо перейти по указанному месту в программе и выполнять команды, начиная с этого места. Когда выполнение этих инструкций будет закончено, программист должен использовать следующую инструкцию goto, чтобы перенаправить компьютер для выполнения дальнейших команд.

Программирование без инструкции goto называют структурированным, поскольку в таких случаях программа состоит из отдельных модулей, назначение которых, как правило, понятно как программистам, так и пользователям. Применение инструкции goto нарушает структуру программного кода и требует от программиста тщательной документации всех мест применения инструкции, что, тем не менее, не спасает от ошибок.

В 1968 г. абсолютное большинство программ было линейными. Такие программы ещё называли "спагетти"

***Прим. В. Шипкова: в нашей стране более распространено название "макароны".**

Для этого стиля свойственно частое применение инструкций goto и отсутствие разбиения на модули. Впрочем, уже тогда использовались функции и подпрограммы, дальнейшее развитие которых привело к возникновению структурированных языков программирования. (Тему функций мы подробнее раскроем в следующих главах.)

В 1968 г. я писал программы на COBOL, и это были типичные представители "спагетти". Хитросплетения кода были ужасно



трудными для чтения и понимания даже автором, а прочесть все это постороннему было вообще невозможно. Единичные структурированные программные конструкции имели ограниченное применение. Они были трудны в использовании, а достать подробную документацию на эти модули было почти невозможно. Единственным методом описания программы была блок-схема, графически представляющая логику программы и взаимосвязи её частей. В диаграмме использовались специальные символы, обозначающие ввод и вывод данных, выбор условий и другие операции. Для облегчения работы программистов были разработаны программы, которые считывали коды других программ и создавали по ним блок-схемы. В идеале предполагалось, что работа программиста должна начинаться с создания блок-схемы, которой он затем должен придерживаться при написании программы. Но в реальной жизни такого почти никогда не бывало. Исходя из личного опыта должен признать, что инструкция `goto` действительно провоцирует программистов к нарушению логической структуры, что приводит к невообразимому хаосу.

COBOL является одним из самых ранних языков программирования. Он помог популяризировать применение компьютеров в эру господства мэйнфреймов – огромных вычислительных машин, таких как IBM 360. Данный язык был разработан адмиралом ВМФ США Грейс Хоппером (Grace Hopper) и её помощниками. Адмирал Хоппер известен своим высказыванием: "Легче получить прощение, чем разрешение". COBOL был примечателен тем, что программы писались "человеческим" языком вместо общепринятого машинного, который понимали только компьютер и профессиональные программисты. Команды COBOL легко было читать и понимать. По сегодняшним меркам COBOL чрезмерно многословен. Я хотел было включить программу "Hello, World", написанную на COBOL, но она оказалась слишком длинной.

Бенджамин Хорф (Benjamin Whorf) и Ноам Шомски (Noam Chomsky) уверены, что язык формирует структуру сознания. Мыслить можно только словами. Действительно, если в Вашем языке отсутствует будущее время, то трудно или даже невозможно размышлять о будущих событиях. Компьютерные языки программирования являются прекрасным подтверждением этой точки зрения. Если в компьютерном языке нет "слова" или метода для выражения того, что Вы хотите сделать, то просто невозможно найти никакое решение. Первые языки программирования представляли собой монолитные конструкции, в которые невозможно было ничего добавить или изменить. Доступ к коду первоисточника и разработчикам языка был ограничен ведомственными интересами компаний-изготовителей. Обычный программист мог использовать – только те средства

программирования, которые уже были в языке, без права что-либо изменять.

Со временем расширяемость языка программирования стало целью, к которой стремились многие разработчики. Когда Guido работал над Python, он приложил массу усилий, чтобы максимально упростить добавление новых средств и модулей в среду программирования. Он отказался также от инструкции `goto`, полностью сосредоточившись на концепции структурированного объектно-ориентированного программирования. Первые теоретики программного конструирования утверждали, что первостепенное значение имеют данные, передаваемые программе. Был провозглашен тезис: "определите структуру данных, и методы, выбранные для их обработки, сами собой станут очевидными". Позже в программировании всё больше внимания стали уделять объектам — программным единицам, которые объединяют как структурированные данные, так и методы их обработки. Guido сумел упростить применение объектов в Python, причём по сравнению с большинством других языков в Python их применение стало значительно проще. Некоторые языки имплантируют объекты в первоначальное линейное ядро, но Python с самого начала был нацелен на объектно-ориентированное программирование. Мы потратим несколько часов на изучение концепций и приемов объектно-ориентированного программирования во второй трети данной книги. Вы на собственном опыте убедитесь, что как только у Вас в голове прояснится понятие объекта, так сразу исчезнут все проблемы с программированием объектов в Python.

Сильные и слабые стороны Python

Python — превосходный язык для реализации многих целей. Python можно использовать для написания сценариев оболочки. При этом часто удаётся значительно сократить число строк в коде и упростить программу. Многие программисты на C++ используют Python для создания прототипов программ. Как уже говорилось, в Python проще придерживаться концепций объектно-ориентированного программирования, а код его более компактный и пишется быстрее. Поэтому бывает полезно написать прототип программы на Python, а затем, если нужно, переписать программу или некоторые её блоки, используя другой язык программирования. Экономия времени и повышение эффективности программирования иногда просто поражает. Создание прототипов в Python тем более удобно, так как этот язык поддерживает интерфейс GUI (Graphical User Interface — графический интерфейс пользователя). Мне приходилось слышать от программистов, которые регулярно использовали Python для создания прототипов, что в результате их проекты

получались более качественными, поскольку Python помогал находить чёткие и элегантные решения. Даже если конечная версия программы должна была быть написанной на C++, применение Python на промежуточном этапе ничуть не замедляло выполнение проекта. Напротив, значительно сокращалось время, затраченное на отладку программы.

В то же время есть области (их немного), в которых Python не может блеснуть своими возможностями. Таковым является анализ больших текстовых файлов с использованием ключевых слов и словосочетаний. Эту же задачу средствами Perl обычно удаётся решить эффективнее, чем с помощью Python, а программа будет работать быстрее. Хотя и в Python существуют довольно мощные средства для решения подобных задач, все же общепринято, что Perl подходит лучше для этих целей.

***Прим. В. Шипкова: на просторах интернета лежит сравнительный анализ скоростных характеристик Питона, Перла и Явы. На сегодняшний день Питон не уступает Перлу по скорости работы с текстом. Общее же превосходство над Перлом оказывается от 2 до 4 раз. Над Явой – от 1,5 до 8 раз!!!**

В то же время простота и быстрота написания программ на Python часто компенсируют потерю в скорости выполнения приложения. Часто критическим оказывается именно время, необходимое для реализации идеи, так как ложка дорога к обеду. Python лучше, чем любой другой язык, соответствует термину "выполняемой идеи" – концепции идеального языка программирования, предложенной Франком Стаяно (Frank Stajano). Суть этой концепции состоит в том, что язык должен быть настолько прост, что программист должен иметь возможность реализовать идеи с ходу, по мере их возникновения. С помощью Python можно не только быстро реализовать свои замыслы, но также анализировать их, делать чётче и понятнее прежде всего для самого себя, что обычно позволяет значительно сократить время на разработку эффективного проекта.

Программы, которые должны работать в режиме реального времени, – не лучшая область применения Python. Интерпретируемые языки вообще слишком медленны для тех случаев, когда необходим мгновенный отклик от выполняемой программы. Python окажется плохим выбором для реализации средств голосовой почты или для коммутации сотни телефонных линий. Впрочем, существуют средства, позволяющие ускорить выполнение программ на Python. Во-первых, наиболее ответственные модули приложения могут быть написаны на

языке низкого уровня, например на С. Другие модули, реализованные на Python, смогут обращаться к С-блокам точно так же, как Python использует собственные встроенные модули. Так, если обработку строк текста написать на С, а все остальные блоки – на Python, то такое приложение будет работать достаточно быстро. Это означает, что Python является расширяемым языком, позволяющим эффективно использовать в приложениях программные блоки, реализованные средствами языков низкого уровня. В свою очередь, низкоуровневые языки могут импортировать в свой программный код модули, написанные на Python. Поэтому, разрабатывая проект на языке С, Вы можете использовать свои наработки на языке Python без необходимости переписывать их. А иногда бывает даже полезно целенаправленно написать часть программного кода на Python, чтобы совместить в приложении сильные стороны низкоуровневого языка программирования с простотой и эффективностью программирования на Python. Кроме того, благодаря расширяемости и импортируемости языка Python его можно использовать для создания связующих модулей, так как с его помощью легко можно связать в одно приложение блоки, написанные на разных языках программирования.

Резюме

В первой главе Вы узнали, что такое программирование, для чего оно может Вам понадобиться и почему Python идеально подойдёт в качестве первого языка программирования. Вы познакомились с краткой историей этого языка и некоторыми историческими данными о языках программирования вообще. Вам также была представлена фундаментальная идея информатики: от выбора языка зависит то, насколько успешно Вы сможете решить поставленную задачу и какие средства идеально подойдут для этого решения. В заключение мы рассмотрели некоторые задачи, для решения которых Python подходит идеально, а также области, где этот язык практически не применим.

Практикум

Вопросы и ответы

Является ли Python переносимым языком?

Можно сделать так, что Python будет работать на любой платформе. Его предварительно откомпилированные двоичные коды доступны для большинства операционных систем. Единственной из когорты ведущих операционных систем, которая не поддерживает его, является NetWare. Но,

насколько мне известно, есть люди, которые работают над устранением этой проблемы.

Могу ли я использовать Python для программирования интерфейса общего шлюза?

Да. Для этого всего лишь необходимо, чтобы на Web-сервере был установлен Python. Возможно, эта идея не понравится администратору Web-сервера, но попробуйте его убедить в том, что ещё один язык программирования CGI никогда не будет лишним.

Безопасен ли Python для программирования CGI?

Можно сказать, что Python в этой области более безопасен, чем Perl, но с точки зрения безопасности идеальным решением, безусловно, будет Java.

***Прим. В. Шипкова: в 2004-2005 годах в Яве было обнаружено несколько уязвимостей, которые оказались весьма серьёзными. Насколько известно мне, таких уязвимостей в Питоне нет до сих пор.**

Подходит ли Python для создания звездоносных приложений, которые прославят авторов в веках и станут популярными во всем мире?

Возможно, для реализации наполеоновских целей — это не самый подходящий язык, но зато он идеален для решения многих рутинных ежедневных задач.

Использует ли Python какая-нибудь из ведущих компаний для разработки своих приложений?

Да, конечно. В число компаний, которые доверились его возможностям, входят NASA, Yahoo, Red Hat, Infoseek и Industrial Light and Magic. Нам известно, что и другие столпы компьютерной индустрии применяют его, но они не склонны афишировать это. Зачем сообщать конкурентам, что высокая эффективность программирования была достигнута благодаря использованию Python. Это не означает, что какие-то компании полностью перешли на программирование на Python. Просто, как уже говорилось в этой главе, использование Python для создания прототипов и разработки отдельных служебных блоков может существенно ускорить выполнение проекта, что, в свою очередь, позволит компании обогнать конкурентов.

Контрольные вопросы

1. Что является веской причиной выбора Python в качестве своего первого языка программирования?
 - а) Мощь, скорость, монолитность.
 - б) Гибкость, расширяемость, импортируемость.
 - в) Elegантность, четкость, простота.
 - г) Возможность программирования в режиме реального времени, мощные средства текстового анализа, схожесть с языком C.
2. Кто первым предположил, что применение в программах инструкции goto разрушает логику и структуру программы?
 - а) Никлаус Вирт.
 - б) Эдсгер Дейкстра.
 - в) Бенджамин Хорф.
 - г) Чарльз Е. Томпсон-младший
3. Кто создал Python?
 - а) Тим Петере.
 - б) Иван ван Лейнингем.
 - в) Гуидо ван Россум.
 - г) Эдсгер Дейкстра.

Ответы

1. б и в. Веским доводом в пользу выбора Python являются его гибкость, расширяемость, импортируемость, элегантность, четкость и простота.
2. б. Эдсгер Дейкстра охарактеризовал инструкцию goto как тлетворную, поскольку она побуждает программистов к превращению своих программ в "комки из спагетти".
3. в. Python был создан Гуидо ван Россумом, хотя от такой славы не отказался бы и Тим Петере, да я и сам был бы не прочь. Но надо отдать должное, что Тим внес огромный вклад в развитие Python (да и вообще он отличный парень!), я же просто счастлив возможности использовать Python в своих целях.

Примеры и задания

Посетите Web-страницу по адресу <http://www.python.org/> - "святилище" всего, что связано с Python. Сразу после того как была написана эта книга, появился ещё один исключительно полезный оперативный ресурс - *The Vaults of Parnassus* (<http://www.vex.net/parnassus>). Это богатый сервер, обладающий широкими поисковыми возможностями,

содержащий массу ссылок на программы, написанные на Python и представленные на серверах по всему миру.

Зарегистрируйтесь в списке рассылки новостей Python или в списке рассылки курсов обучения Python, которые представлены по адресу <http://www.python.org/psa/MailingLists.html>.

Познакомьтесь со статьей Эдсгера Дейкстра об инструкции goto, которую можно найти на Web-странице ACM по адресу <http://www.acm.org/classics/oct95/>.

Составьте список своих рутинных задач, которые Вам приходится выполнять ежедневно и с которыми, как Вам кажется, мог бы справиться Ваш компьютер без Вашего участия. По мере ознакомления с содержанием данной книги возвращайтесь к этому списку и пробуйте применить полученные знания для решения этих проблем. ещё до окончания чтения книги постарайтесь воплотить хотя бы одно из этих решений, экономящих Ваше время. Если Вы написали только одну программу и сумели тем самым сэкономить себе хотя бы несколько минут в неделю, значит, Вы не зря потратились на эту книгу.

2-й час

Интерпретатор Python

В этой главе Вы познакомитесь с интерпретатором Python, узнаете, чем компилируемые языки отличаются от интерпретируемых, что такое IDLE и как запускать это приложение. И самое главное — мы напишем и запустим на выполнение нашу самую первую программу на Python.

Интерпретатор: краткие сведения

Существуют две основные категории языков программирования: *компилируемые* и *интерпретируемые*. Компилируемые программы отличаются своей быстротой, но интерпретируемые приложения, как правило, проще в использовании и отладке. Компилируемые языки отличаются тем, что сначала программист набирает код программы с помощью какого-либо текстового редактора и сохраняет результат в виде файла. После того как текст программы записан в файл, программист запускает специальную программу, называемую компилятором, которая обрабатывает этот сохранённый файл. Компилятор считывает код источника (код источника — это как раз то, что Вы набираете в текстовом редакторе и сохраняете в файле) и

преобразовывает ключевые слова языка (т.е. команды, удобные для чтения и понимания с точки зрения человеческой логики) в инструкции машинного кода, который понятен только компьютеру. После завершения компиляции машинный код в двоичном формате записывается в специальный файл. В DOS и Windows имя этого файла обязательно заканчивается расширением .exe, которое сообщает операционной системе, что этот файл является выполняемым. На машинах с UNIX операционная система предоставляет программисту особую процедуру, которая позволяет отметить как выполняемый файл, содержащий инструкции в машинном коде. В любом случае, чтобы запустить программу на выполнение, пользователь может просто ввести в командной строке соответствующего окна (или интерфейса) имя файла (расширение .exe, если файл его имеет, в DOS и Windows можно даже пропустить). Большая часть всех крупных приложений, таких как текстовые редакторы, Web-браузеры и программы обработки баз данных, созданы именно таким образом.

Редактор — это программа, которая позволяет пользователю вводить в её рабочее окно инструкции, данные и комментарии в виде текста и сохранять в файле набранную текстовую информацию. В качестве примера редактора можно привести программу vi из UNIX и Блокнот из Windows.

***Прим. В. Шипкова: я пользуюсь плагином SynPlus, который является одним из многочисленных плагинов к файловому менеджеру Total Commander.**

В отличие от компилируемых языков программирования, с помощью интерпретируемых языков нельзя создавать выполняемые файлы. Программы, написанные на языке интерпретатора, могут выполняться одним из двух способов: в интерактивном (диалоговом) режиме или в пакетном. Выполнение программы в интерактивном режиме часто доставляет определённые неудобства, поскольку программист сначала должен запустить программу, называемую интерпретатором (для Python эта программа запускается файлом python.exe или просто python). Затем программист вводит команды непосредственно в интерпретатор. Недостаток этого метода состоит в том, что программы, набранные таким образом, перестают существовать сразу после прекращения работы интерпретатора. В пакетном режиме программист сначала записывает текст программы в файл (аналогично тому, как это происходит для компилируемых языков), а затем даёт указание интерпретатору выполнить этот файл. Вот пример запуска программы на языке Python в пакетном режиме (как для UNIX, так и для DOS) —


```
c:\>python hello.py
```

Эта командная строка сообщает операционной системе, что сначала необходимо запустить на выполнение программу `python`, а после того как Python запустится, в память машины автоматически считывается файл `hello.py` и выполняются содержащиеся в нем команды.

Интерпретатор считывает файл, но вместо того чтобы сразу же преобразовать всю программу в машинный код, он выполняет каждую командную строку файла по мере их прочтения. Удобство, по сравнению с компиляцией, состоит в том, что при обнаружении ошибки интерпретатор сразу же сообщает о ней программисту. Программист может немедленно исправить её и продолжить выполнение программы. Это свойство предоставляет огромное преимущество при отладке программы, так как позволяет исправлять ошибки по мере их обнаружения, не затрачивая время на компиляцию каждой новой версии программы.

Диалоговый режим очень полезен и весьма эффективен в тех случаях, когда Вы испытываете какой-то новый метод или оригинальный нестандартный приём, с которым Вы только что познакомились, или когда хотите задать интерпретатору несколько простых операций, наблюдение за выполнением которых поможет Вам лучше понять принципы данного языка программирования. Изучая материал первых нескольких глав, мы будем работать в основном в диалоговом режиме, потому что на этом этапе Вам необходимо сосредоточиться на базовых инструкциях языка Python. На практических занятиях Вы будете постигать основные подходы программирования, которые окажутся полезными при разработке ваших собственных проектов. Таким образом, цель наших первых занятий — освоение основ, а не создание прикладных программ, которые имело бы смысл сохранять.

Прежде чем Вы сможете запустить интерпретатор Python, необходимо получить и установить его, если он ещё не установлен на Вашем компьютере. К примеру, система Red Hat Linux устанавливает Python в процессе инсталляции операционной системы Linux, так что Вы сразу же можете приступать к работе. При использовании дистрибутивов Windows и других версий Linux или UNIX установку Python следует выполнить отдельно (в приложении Г Вы найдёте ссылки на Web-страницы, предоставляющие как саму программу для установки, так и полезную информацию о том, как правильно установить и использовать Python.) Если Ваш поставщик уже инсталлировал Python на Вашем компьютере, удостоверьтесь, что установлен также редактор IDLE. IDLE —




это специальная программа текстового редактирования, которую Guido van Rossum полностью написал на Python. Мы будем использовать IDLE для редактирования и выполнения примеров программ на протяжении всей книги. Если эта программа у Вас ещё не установлена, опять-таки обратитесь к узлу <http://www.python.org/>. Обратите внимание, что при инсталляции версии Python для Windows редактор IDLE устанавливается автоматически.

Если Вы устанавливаете версию Python для Windows, я настоятельно рекомендую обратить внимание на одну деталь – укажите инсталляционной программе поместить Python в папку под именем Python в корневом каталоге. В процессе установки инсталлятор по умолчанию предлагает папку C:\Program Files\Python, но я считаю, что лучше выбрать C:\Python. Дело в том, что наличие пробелов в именах каталогов может в дальнейшем привести к трудностям при работе с некоторыми старыми программами, написанными на Python, в которых не предполагалось наличие пробелов в именах каталогов. Впрочем, что касается примеров программ, предложенных в этой книге, то они должны выполняться без проблем и в том случае, если Вы разместите Python в папке, предложенной по умолчанию.

***Прим. В. Шипкова: как правило, сейчас это указание несущественно. Я уже давно не сталкивался с подобным. Более реальна проблема, из-за разного способа обозначений пути в *nix и Windows. В любом случае – я не ставлю ни одну программу на диск, где стоит Windows. Уж очень капризная система.**

Независимо от того, работаете Вы под управлением операционной системы Windows, Macintosh или UNIX/Linux, необходимо убедиться, установлен ли модуль Tcl/Tk. В случае с Windows все просто – ответьте Yes (Да), когда программа установки спросит, хотите ли Вы установить модуль Tcl/Tk. У пользователей UNIX возникнут определённые проблемы. UNIX вообще предполагает (или даже требует), чтобы его пользователи больше знали о своём компьютере, чем пользователи Windows или Macintosh. Возможно, Вам даже придётся прибегнуть к помощи системного администратора и уговорить его помочь Вам. Ссылки на серверы, с которых можно загрузить Tcl/Tk для UNIX, находятся по адресу <http://www.python.org/download/>. Вы также можете найти инсталляционные пакеты для Linux в двоичном коде на сервере <http://www.andrich.net/python/>. Это необходимо сделать, так как для полного усвоения материала этой книги Вам в любом случае понадобится модуль Tcl/Tk. (Работе с

этим модулем полностью посвящена заключительная часть книги.)



Обратите внимание, что для работы с Python 1.5.2 требуется версия Tcl/Tk 8.0. Модуль Tcl/Tk 8.1 не подойдет, или, по крайней мере, Вам придётся самостоятельно выполнить огромный кусок работы, неподъёмный для новичка. Программа установки Python для Windows наверняка выберет правильную версию модуля, если только Вы зададите установку Tcl/Tk. Но в случае с UNIX Вам, возможно, придётся загрузить, скомпилировать и установить этот пакет самостоятельно. Так как эта тема выходит за рамки данной книги, руководство по установке Вы найдёте на Web-страницах, предложенных разделе "Примеры и задания" в конце главы.

Для пользователей Windows я рекомендую дополнительно загрузить и установить ещё два модуля. Они не понадобятся Вам при работе с данной книгой, но станут хорошим подспорьем для дальнейшей работы по мере того, как Вы будете сталкиваться со всё новыми и новыми задачами программирования в Python. Вот эти модули— Win32Api Extensions и PythonWin. Модуль Win32Api расширяет ваши возможности и позволяет работать с теми же элементами операционных систем семейства Windows, к которым имеют доступ программисты, пишущие на компилируемых языках типа C. Модуль PythonWin предоставляет альтернативный редактор и среду программирования, которые отличаются от использованных в этой книге. Некоторые пользователи предпочитают именно данный редактор, а не IDLE. Это дело вкуса, для целей данной книги IDLE подходит идеально.

***Прим. В. Шипкова: этот модуль у меня есть. Но, к сожалению, запустить его не удаётся. Впрочем, я не сильно пытался разобраться в причинах такого поведения.**

Интерактивность и окружение

Итак, Python успешно установлен на вашей машине. Но чтобы работа с ним была плодотворной, необходимо приложить ещё немного усилий. Обычно большинство инсталляционных процедур учитывает эти вопросы автоматически. Например, в процессе инсталляции под Windows изменяется содержимое системного реестра, куда добавляются параметры настройки, необходимые для того, чтобы в процессе своей работы программа Python могла найти нужные библиотеки методов и функций. Если Python достался Вам уже предварительно установленным (и настроенным) в операционной системе Linux, информация о размещении необходимых библиотек указана в специальном

файле site.py. В обоих случаях Вам не понадобится собственноручно вносить какие-либо изменения.

В более ранних версиях Python Вам, скорее всего, пришлось бы устанавливать параметры системной переменной PYTHONPATH. Но с появлением Python 1.5.2 необходимость в этом отпала. Правильно установленный Python знает, где искать необходимые модули (с которыми Вы познакомитесь подробнее немного позже), и загрузит их без всяких проблем. Необходимость в установке PYTHONPATH может возникнуть только в особых случаях, например, если Вы хотите загрузить модули, о наличии которых не должны знать остальные сотрудники, работающие на Вашем компьютере. Из всех операционных систем больше всего работы потребуется от пользователей Windows NT. Вам необходимо будет добавить путь к программам Python и Tcl в системные свойства.

***Прим. В. Шипкова: в версии начиная от 2.2 ничего делать не нужно - проверено на собственной шкурке.**

Это можно выполнить следующим образом: последовательно выберите команды
Start (Пуск)>Settings
(Настройка)>Control Panel (Панель управления),
а затем дважды щелкните на пиктограмме System (Система). После того как появится окно System, щелкните на вкладке Environment (Среда). Вы увидите страницу свойств, примерный вид которой показан на рис. 2.1 (так это окно выглядит на моём домашнем компьютере).

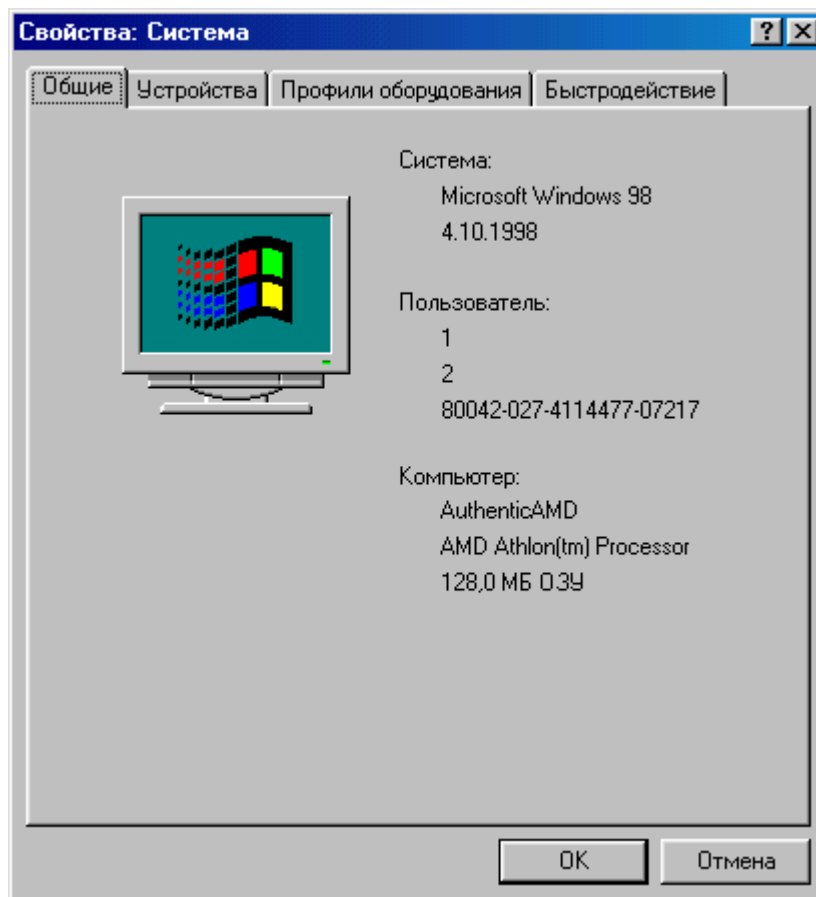


Рис. 2.1. Окно Свойство системы Windows 98

Обратите внимание, что в колонке Value (Значение) указано местоположение интерпретатора Python и библиотеки Tcl. После того как Вы добавите эти папки в пользовательские настройки вкладки Environment и щелкнете на кнопке Apply (Применить), Ваш Python должен будет выполняться без каких-либо проблем. Чтобы внесенные Вами изменения обрели силу, необходимо закрыть, а затем повторно открыть все выполнявшиеся до этого командные окна программ.

Если Вы введёте команду `python`, а в ответ ваша операционная система пожалуется, что во всей вселенной не существует ничего подобного, значит, Вам необходимо изменить свою системную переменную `PATH`. Сначала проверьте, в той ли папке установлен Python, где Вы предполагали. Если у Вас не хватает знаний или прав доступа, чтобы самостоятельно найти папку, где был установлен Python, обратитесь к своему системному администратору. Установив местонахождение Python на своём компьютере (предположим, это папка `C:\Python\`), проверьте значение переменной `PATH`. Вам повезло, если этот путь в списке отсутствует. В таком случае не составит труда его добавить. Если же путь был представлен в переменной `PATH`, значит, ошибка в чем-то другом и Вам потребуется квалифицированная помощь для выяснения её причины.

Чтобы установить размещение Python в каталоге систем UNIX/Linux, необходимо ввести в командной строке оболочки что-то вроде `PATH=$PATH:/usr/local/bin; export PATH`. Или, что даже ещё лучше, добавьте эту строку в конфигурационный файл запуска оболочки. Этот файл может называться `.profile`, `.bashrc` или `.cshrc`, в зависимости от того, какую оболочку Вы используете. В любом случае этот файл должен находиться в корневом каталоге. Откройте этот файл с помощью текстового редактора и добавьте соответствующую информацию о местоположении в строку, где уже установлены другие параметры вашей переменной `PATH`. Если Вы не вполне понимаете, о чём идёт речь, лучше прибегнуть к помощи специалиста. Но в действительности всё не так уж и сложно. Просто внимательно прочтите установки, которые уже имеются в Вашем файле конфигурации, и их контекст подскажет Вам, что именно необходимо изменить.

Если местоположение Python описывается, к примеру, такой строкой, как `c:\python\python.exe`, то Вам необходимо ввести в окне DOS (в строке приглашения на ввод команды) следующую запись: `PATH %PATH%; C:\Python; c:\Python\Tcl; c:\Python\Tcl\bin`. Если Вы работаете в Windows 95/98, добавьте упомянутую строку где-нибудь в конце Вашего файла `AUTOEXEC.BAT` и перезапустите систему. На рис 2.2 изображено внесение изменения в файл `AUTOEXEC.BAT` с помощью программы Блокнот.

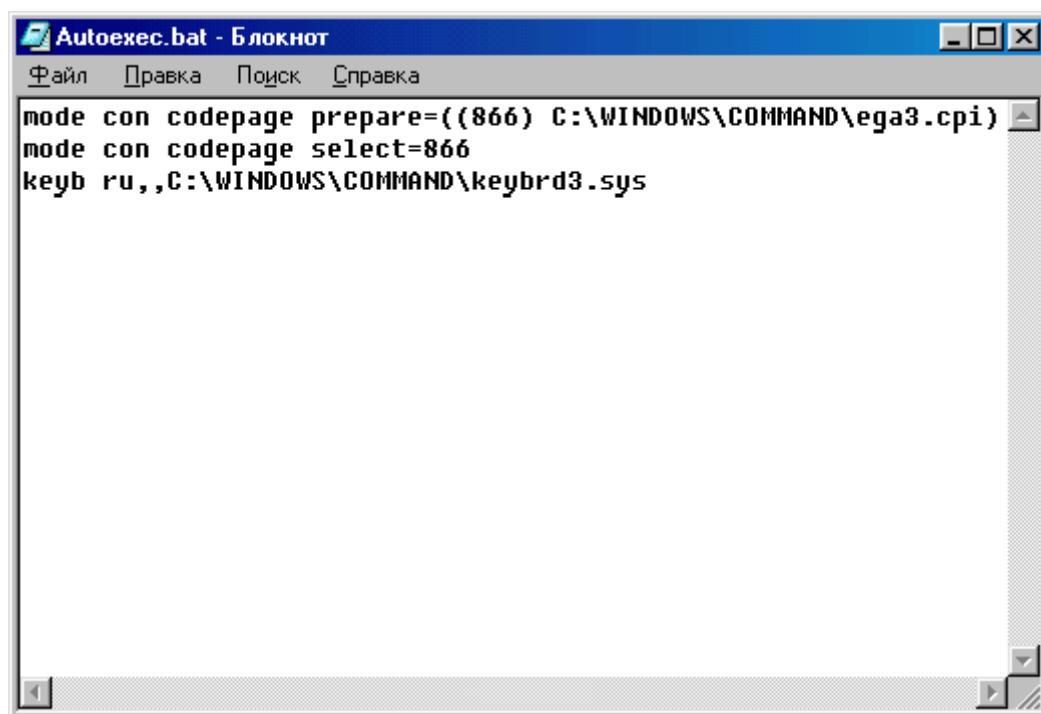


Рис. 2.2. Изменение файла `AUTOEXEC.BAT`

Наконец, Вы можете запустить Python, введя `python` в командной строке Windows или UNIX и нажав <Enter>. Откроется окно, показанное на рис 2.3.

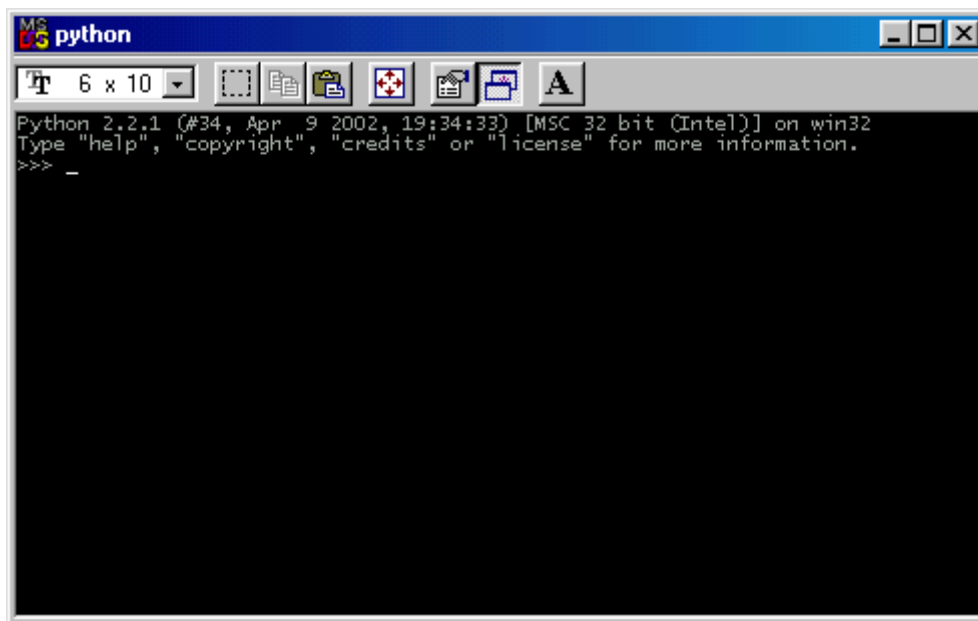


Рис. 2.3. Такое окно должно открыться после ввода команды `python`

Символы `>>>` называются приглашением Python. Приглашение сообщает, что интерпретатор Python готов принять от Вас команду. Если ещё не дрожат руки от волнения, можете посмотреть, что получится, когда Вы начнете вводить команды. Попробуйте ввести "Hello, World" (убедитесь в том, что Вы не забыли ввести и кавычки) и нажмите клавишу <Enter>. На рис. 2.4 представлено ещё несколько примеров, которые стоит попробовать выполнить.

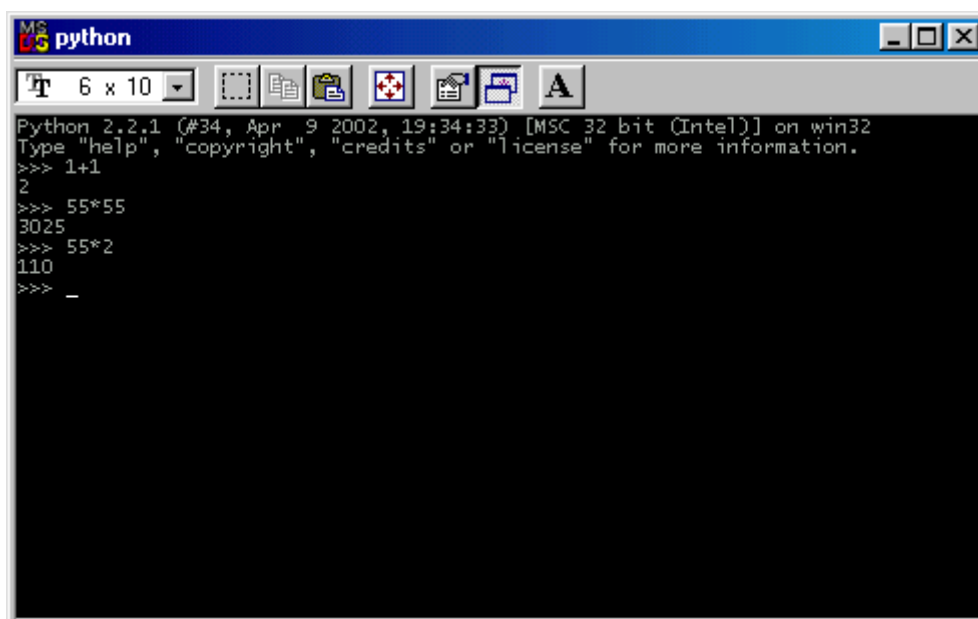


Рис.2.4. Вот так вводятся и выполняются команды в интерпретаторе

Чтобы выйти из интерпретатора (т.е. завершить работу с ним), в Windows воспользуйтесь комбинацией клавиш <Ctrl+Z>. Для UNIX и при работе с IDLE комбинация другая – <Ctrl+D>. В обоих случаях для ввода этих команд необходимо, удерживая нажатой клавишу <Ctrl>, одновременно нажать клавишу <Z> или <D>. Есть одно общее правило – сначала надо научиться тому, как выйти из любой программы (системы, дома, ситуации и т.д.), прежде чем браться за что-нибудь ещё.

Выполнение сценариев

Хотя ввод инструкций в интерпретаторе может показаться делом забавным, простым и даже полезным для обучения разным функциям, все это не стоит и ломаного гроша, если Вы не можете сохранять результаты своей работы. В обычном интерпретаторе, который работает в режиме командной строки (в окне DOS на компьютерах типа PC или окне эмулятора терминала на UNIX, и т.д.), нет возможности сохранить плоды ваших трудов. Работа, ограниченная только вводом команд в интерактивном режиме, подобна бросанию денег в море: после этого Вы чувствуете невероятный эмоциональный подъём, но уже никогда не можете вернуть их обратно.

***Прим. В. Шипкова: я бы сказал, более того – если результат работы нельзя было бы сохранять, то труд программиста потерял бы всякий смысл. К счастью, существует множество способов сохранять данные. %)**

Все же в этой, казалось бы, безвыходной ситуации имеется решение. Можно воспользоваться текстовым редактором и с его помощью набрать текст программы, после чего сохранить его в файле, который должен заканчиваться расширением .py. (Как уже упоминалось выше, в Windows Вы можете использовать для написания кода приложение "Блокнот" или любой другой текстовый редактор, которому отдаёте предпочтение.) Кстати, Python не требует, чтобы файлы с кодом обязательно заканчивались расширением .py, но я настоятельно рекомендую поступать именно так, особенно если Вы работаете в Windows. Гарантирую, работа станет значительно легче.

Независимо от типа платформы, с которой Вы работаете, метод запуска созданных таким образом документов (а чтобы окружающие считали Вас крутым специалистом, смело называйте их сценариями) одинаков. Необходимо всего лишь ввести с клавиатуры в приглашении операционной системы (не важно, какой) следующую команду: `python script.py`

В следующем разделе мы и займёмся этим вплотную.

Привет, Python!

Хотя лично я для написания своих сценариев на языке Python предпочитаю использовать редактор vi, Вам лучше всего использовать тот редактор, в котором Вы чувствуете себя наиболее комфортно. На первоначальном этапе этим требованиям полностью соответствует приложение "Блокнот", которое входит в пакет стандартных программ Windows. Но позже Вы почувствуете, что ему не хватает многих полезных свойств, например автоматической установки отступа, что со временем начнет Вас раздражать.

Новый термин

Код — это строки читабельных команд, записанные на определённом языке программирования.

Раскройте окно своего редактора и введите следующий код:

```
print "Hello, World!"  
print "Goodbye, World!"
```

(Убедитесь, что эти две строки начинаются с крайней левой позиции Вашего текстового документа. Перед инструкцией `print` не должно быть ни одного пробела.) Сохраните эти строки в текстовом файле под названием `helloworld.py`, откройте окно DOS (или другое окно терминала) и сделайте текущим тот каталог, где Вы сохранили файл. Внимание! Теперь Вы готовы запустить свою первую настоящую программу на языке Python.

Новый термин

Под термином пробел подразумеваются знаки табуляции, пробела и символы перевода каретки и разрыва строки. В Python очень важно не забывать о пробелах, потому что символы пробелов или табуляции, стоящие в начале строки, указывают на структурный уровень данной строки. Все строки программы с определённым отступом рассматриваются интерпретатором как единый программный блок. Другие языки для указания программного блока используют специальные символы или ключевые слова. В этом случае отступы, задаваемые пробелами, служат лишь для того, чтобы улучшить читабельность кода. В других языках специальные символы используются ещё для указания конца выражения, что позволяет разбивать выражение на несколько строк. В Python

символ разрыва строки указывает одновременно и конец выражения.

Новый термин

Ключевое слово – это слово, которое зарезервировано в данном языке программирования для выполнения какой-либо функции. Например, `print` – ключевое слово в Python. Совершенно недопустимо использовать в программах ключевые слова в качестве имён переменных или функций (с переменными и функциями Вы познакомитесь несколько позже).

Введите в окне терминала строку `python helloworld.py` и посмотрите, что произойдет. Надеюсь, Вы увидите на своём экране то же, что изображено на рис. 2.5.

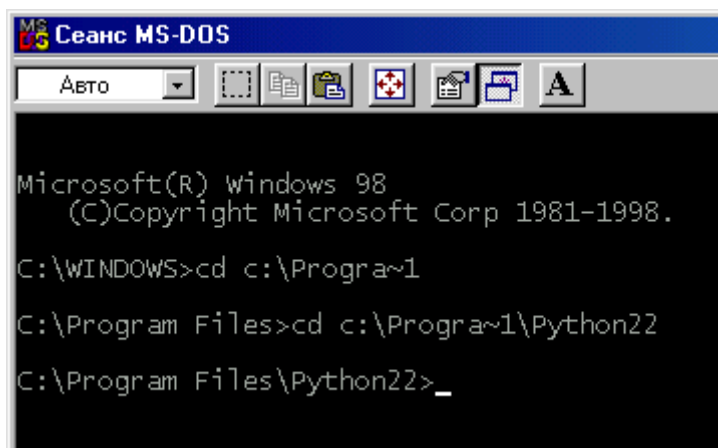
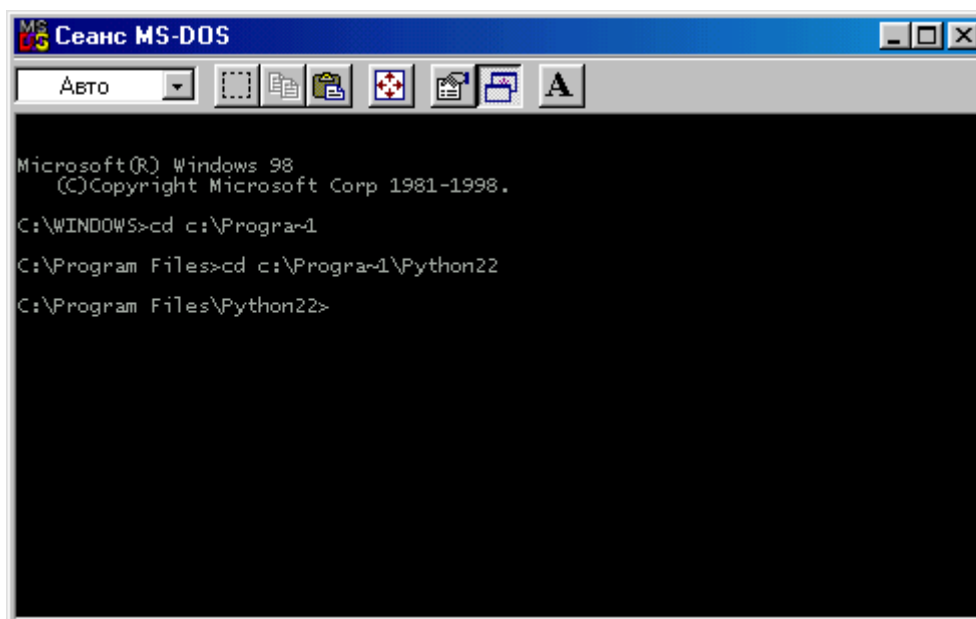


Рис. 2.5. Привет, Python!

Вы только что выполнили свою самую первую программу на языке Python. По традиции обучения любому языку программирования первая программа приветствует мир: "Hello,

World". Пока что это и последняя программа на ближайшее время. В следующей главе мы сосредоточимся на усвоении основных математических операторов в Python и воспользуемся для этого возможностью работы с интерпретатором в диалоговом режиме. Но вместо окна терминала мы будем использовать более совершенный программный продукт под названием IDLE.

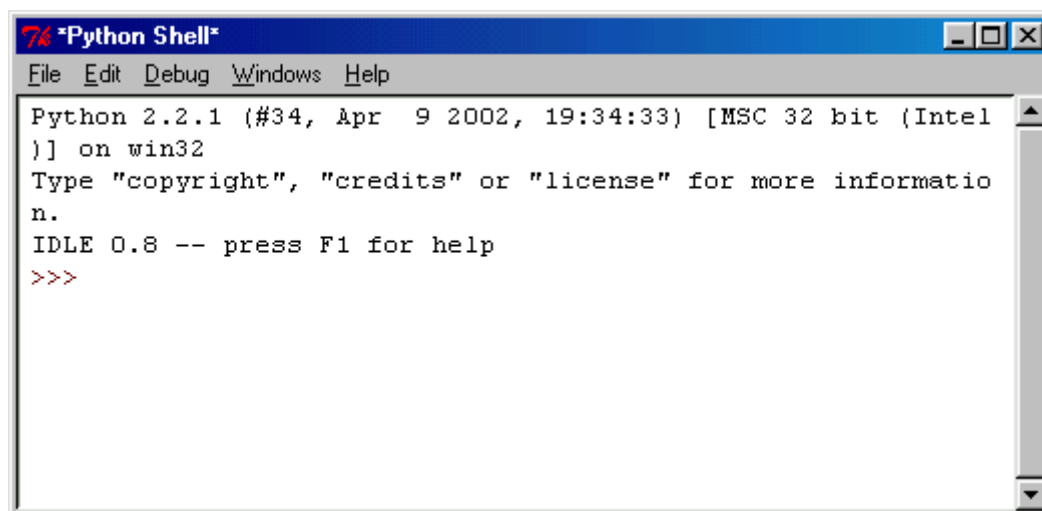
IDLE — аббревиатура от Integrated DeveLopment Environment (интегрированная среда разработки). Эта программа была написана создателем Python, Гуидо ван Россумом, причём полностью на языке Python с использованием компонента графического интерфейса пользователя Python под названием *tkinter*. Мы познакомимся с *tkinter* более подробно в завершающей трети данной книги, а пока работа с IDLE даст Вам общее представление о разновидностях проектов, которые можно реализовать с помощью *tkinter*.

Если Вы уже изменили значение переменной `PATH` таким образом, что в ней указаны каталоги Python и Tel, самый простой способ обеспечить постоянную доступность IDLE — это создать ярлык к нему на рабочем столе своего ПК. Это очень просто осуществить в Windows и не слишком трудно в Red Hat Linux 6.0, но это достаточно сложно в других версиях UNIX. Я опишу, как создать ярлык в Windows, но пользователям Linux и UNIX я рекомендую обратиться к имеющейся у них документации, чтобы уточнить, как это лучше сделать на их платформе (если это вообще возможно). В противном случае для его запуска необходимо ввести с клавиатуры `idle` в окне эмулятора терминала.

Первое, что необходимо сделать, — это найти, где располагается программа IDLE, а для этого Вы должны знать, где находится Python. Но этот вопрос уже должен решаться просто, поскольку каталог, в котором находится Python, должен быть прописан в ваших системных параметрах. На всех моих компьютерах с Windows Python находится в каталоге `c:\Python`. Если во время установки Python Вы выбрали ту же папку, то откройте её в окне приложения Проводник. В папке Python Вы должны найти несколько вложенных папок, одна из которых называется `Tools`. Раскройте эту папку. Она должна содержать другую папку под именем `idle`. Внутри папки `idle` есть файл `idle.py`, а сразу за ним должен находиться файл `idle.pyw`, который и является целью наших поисков. Щелкните правой кнопкой мыши на пиктограмме `idle.pyw` и из появившегося контекстного меню выберите опцию Создать ярлык. Windows создаст в текущем каталоге новый ярлык под именем `Shortcut to idle.pyw` (Ярлык для `idle.pyw`). Перетащите эту пиктограмму на рабочий стол Windows. При

желании можно изменить название ярлыка и даже саму пиктограмму (хотя лично мне нравится этот маленький зеленый питончик, по-моему, он украсит Ваш рабочий стол) . Существует альтернативный способ создания ярлыка для быстрого доступа к приложению IDLE. Щелкните в главном меню кнопки Пуск на опции Настройка и выберите в раскрывшемся подменю пункт Панель задач и меню "Пуск"... . В появившемся окне перейдите на вкладку Настройка меню и щелкните на кнопке Дополнительно... . В папке Программы найдите вложенную папку Python 1.5 и выберите её. На правой панели окна Обзор появятся четыре пиктограммы, одна из которых подписана как IDLE (Python GUI) . Щелкните правой кнопкой мыши на этой пиктограмме и выберите команду Создать ярлык. Перетащите созданный ярлык на рабочий стол Windows и при желании переименуйте и подберите пиктограмму на свой вкус. После того как Вы будете полностью удовлетворены внешним видом и положением на рабочем столе Windows ярлыка программы IDLE, двойным щелчком мыши запустите эту программу на выполнение. На экране должно появиться главное окно, внешний вид которого показан на рис. 2.6.

***Прим. В. Шипкова: сейчас в самом ходу версия IDLE 1.1.1 – разумеется, идёт в комплекте с Питоном.**



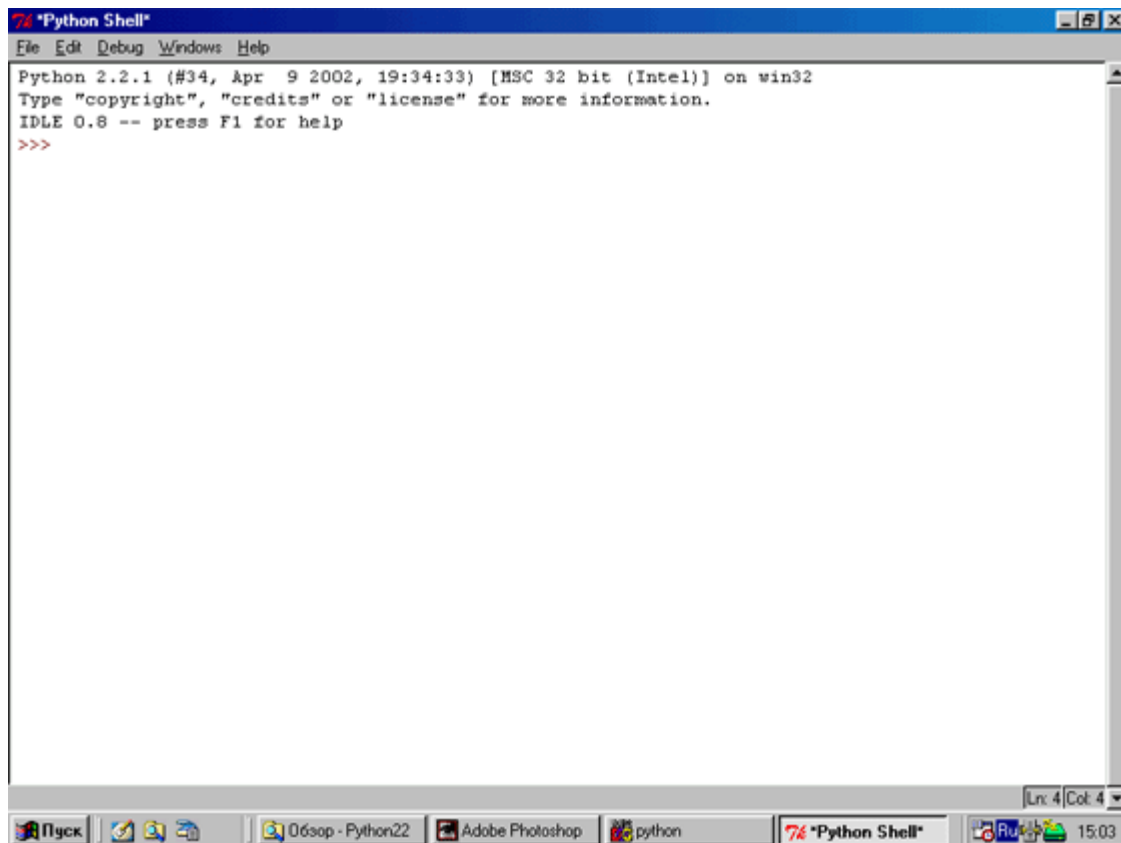


Рис. 2.6. Главное окно программы IDLE

Из программы IDLE можно выйти тремя способами: выбрать в меню File (Файл) опцию Exit (Выход), или ввести с клавиатуры комбинацию клавиш <Ctrl+D> в поле главного окна, или нажать комбинацию <Ctrl+Q> в любом активном окне IDLE. Попробуйте выйти из программы одним из приведенных способов, затем снова запустите IDLE, чтобы испытать другие методы выхода. Способ, который понравится Вам больше всего, применяйте и в дальнейшем. Теперь Вы готовы продолжить работу и перейти к более сложным вопросам. В следующей главе Вы узнаете, как выполнять в Python основные математические действия, а поможет Вам в этом программа IDLE.

Резюме

В течение второго часа обучения Вы узнали об основных различиях между компилируемыми и интерпретируемыми языками программирования. Вы научились устанавливать Python и Tcl. Чтобы Python заработал в вашей системе, иногда приходится вручную указать путь к его каталогу в параметрах. После завершения установки Python можно запустить из окна DOS или окна эмулятора терминала. Важно также, что Вы узнали, как завершить работу интерпретатора. Вы узнали, как выполнять простые сценарии, как создать на рабочем столе ярлык приложения IDLE и как запускать эту программу. У Вас также не должно возникнуть проблем с выходом из программы IDLE.

Практикум

Вопросы и ответы

Какие платформы поддерживают IDLE?

UNIX, Windows и Macintosh, поскольку только эти платформы поддерживают работу tkinter.

***Прим. В. Шипкова: и Windows тоже, так как пакет Tcl/Tk ставится вместе с Python. А без него IDLE смысла не имеет.**

Что делать, если IDLE не запускается?

В основе этой проблемы может лежать множество причин, поэтому трудно ответить однозначно. Попробуйте обратиться за помощью к тому, кто, по вашему мнению, лучше разбирается в компьютерах. Если помощь окажется малоэффективной, то лучше всего посетить домашнюю страницу Python по адресу <http://www.python.org/>. Найдите и зарегистрируйтесь в списке рассылки Python Tutor. Этот сервер специально предназначен для оказания помощи новичкам (вроде Вас) в преодолении трудностей программирования на Python.

Могу ли я выполнять программы Python без ввода python script.py?

Да. В UNIX можно воспользоваться утилитой chmod, которая делает файлы сценариев выполняемыми. Введите chmod +x script.py, и эта команда укажет UNIX (или Linux), что данный файл необходимо обрабатывать таким же способом, как и программы. Кроме того, Вам необходимо будет добавить в программе первую строку, которая должна выглядеть примерно следующим образом:

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

Информацию о правильном синтаксисе этой строки можно найти в разделе документации операционной системы, посвящённом работе с выполняемыми файлами. После этого Вы сможете просто вводить script.py в командной строке. Аналогичный метод существует также и для Windows NT, но в этом случае процедура окажется существенно сложнее. Если поискать соответствующие документы на домашней страничке Python (<http://www.python.org/>), то можно найти исчерпывающее руководство о том, как правильно её выполнить.

***Прим. В. Шипкова: подобного рода извращения не потребуются. Так как при установке Python регистрирует расширение файлов .py, .pyw, .pyc - "на себя". Знаки таких файлов становятся как у приложения python.exe, т. е. питон.**

Контрольные вопросы

1. Что такое IDLE? (IDLE переводится с английского как бесполезный, ленивый, праздный.)
 - а) Яблоко, запеченное в тесте (блюдо в южной части Индии) .
 - б) Интегрированная среда интерпретатора, написанная полностью с помощью Python.
 - в) То, во что Вы превращаетесь каждую субботу.
 - г) То, чем все время является Ваш кот.
2. Что происходит, если после ввода команды python в ответ Вы получаете сообщение, в котором говорится, что "The name specified is not recognized as an internal or external command, operable program, or batch file" ("Указанное имя не распознано как внутренняя или внешняя команда, действующая программа или пакетный файл") ?
 - а) Вы неправильно установили параметры пути.
 - б) Кто-то злодейски похитил у Вас Python.
 - в) Компьютер просто немного упрямый, попробуйте повторить ввод команды.
 - г) Вы работаете под управлением Windows, а Билл Гейтс не любит Python.
3. Что представляет собой сочетание символов >>>?
 - а) Магия Вуду.
 - б) Знак "тройная стрелка".
 - в) Приглашение Python, которое сообщает, что Python нетерпеливо ожидает ввода ваших команд.
 - г) Просто мусор (случайные данные) .
4. Что произойдет, если в интерпретаторе Python ввести команду, которую он не распознает?
 - а) Ничего.
 - б) Это приведёт к перезагрузке компьютера.
 - в) Интерпретатор выведет специальное сообщение об ошибке в виде NameError: <слово, которое Вы ввели>. Если Вы ввели два или больше слов, интерпретатор покажет сообщение SyntaxError:.

г) Он заставит Вас ввести эту команду правильно 500 раз.

Ответы

1. б. IDLE – интегрированная среда интерпретатора, написанная полностью на языке Python.
2. а. Если в приглашении командной строки DOS выводится сообщение о том, что DOS не распознала имя команды, значит, Вы неправильно установили параметры пути.
3. в. Символы >>> – это приглашение Python. Убедитесь, что отличаете её от приглашения командной строки в окне DOS, которое распознается DOS как компонент имени.
4. с. Если в интерпретаторе Python ввести команду, которую он не распознает, то на экране появится сообщение об ошибке. Эти сообщения, как правило (но не всегда), довольно информативны. К счастью, Python не перезагрузит Ваш компьютера и даже не завершит свою работу. Однажды я работал на машине, которая требовала периодически запускать специальные диагностические программы, и чтобы войти в этот особый режим диагностики, необходимо было без единой ошибки ввести примерно 15 команд. Причем при их вводе надо было учитывать то, что все диагностические программы имели чрезвычайно придирчивый синтаксис, который включал примерно 30 параметров в командной строке, разделенных запятыми. Если какой-то параметр не использовался, в любом случае надо было ввести запятую, просто чтобы сообщить машине, что данный параметр не используется. Кстати, большинство параметров действительно не использовалось. Если Вы набирали команду и вводили слишком много или слишком мало запятых, диагностический режим прекращался, а машина зависала, вынуждая осуществлять перезагрузку. Как Вы догадываетесь, эта машина не пользовалась особой популярностью.

Примеры и задания

Позэкспериментируйте с интерпретатором, запуская его в окне DOS, а затем с помощью IDLE. Посмотрите, сможете ли Вы найти отличия в том, как ведет себя каждая из программ.

Посетите домашнюю страничку Python (<http://www.python.org/>) и отыщите страницу, содержащую список рассылки. Зарегистрируйтесь в списке рассылки обучающего курса Python. В этом списке можно найти и автора этой книги, а также фамилии многих других людей, сведущих в Python. Мы ведем этот курс именно для того, чтобы отвечать на ваши вопросы и помогать преодолевать трудности в



освоении Python. Быть может, когда-нибудь и Вы займете место среди преподавателей этого курса.

3-й час

Арифметические действия в Python

В этой главе мы изучим основные арифметические действия, которые можно выполнять в Python и IDLE. Хотя они, по сути, предельно простые, но на их основе строятся все сложные математические вычисления, некоторые из которых мы рассмотрим далее в этой книге. Впрочем, чтобы стать хорошим программистом, совершенно необязательно быть доктором математических наук. В большинстве случаев от Вас потребуются лишь знания основ математики на уровне средней школы. Если же Вам приходилось изучать курс высшей математики, значит, Вы подготовлены на все 100%. Так что не бойтесь, смело вперед. Уж, по крайней мере, в этой главе с математикой у Вас проблем не будет, только с программированием.

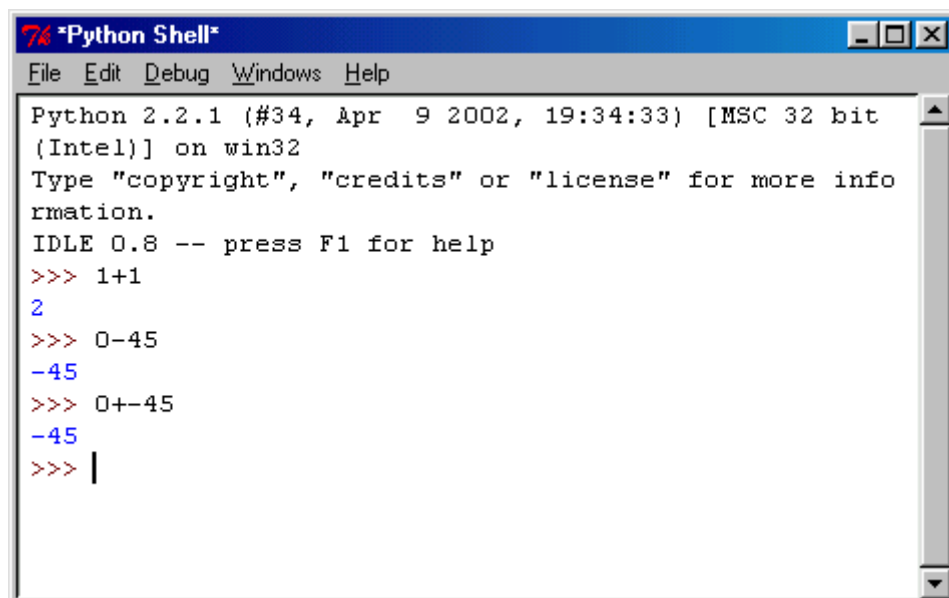
Тем, кто не слишком преуспевал в математике, не следует отчаиваться. В старших классах средней школы я, как и все, изучал алгебру и почти провалил её на экзамене (у меня был откровенно слабый преподаватель, но что ещё хуже, я не уделял этому предмету должного внимания). Много позже, на армейском вводном курсе по электронике я наверстал упущенное и усвоил некоторые из основных методов. Через два года после окончания этого курса я стал его преподавать.

Сложение и вычитание

В любой начальной школе ещё с первого класса изучают операции сложения и вычитания, так как это тот столп, на котором держится вся математика. Я не сомневаюсь, что Вы знаете, как выполнять эти действия. И что особенно приятно, в Python (как, впрочем, и в любом другом языке программирования) все происходит именно так, как Вы и предполагаете.

Выполним несколько упражнений с операциями сложения и вычитания. Запустите интерпретатор Python, введя `python` в приглашении командной строки, а затем в приглашении Python введите `1 + 1`. Если после нажатия клавиши <Enter> Python не ответит Вам результатом 2, значит, у Вас большие проблемы! Теперь попробуйте выполнить более сложные задачи, например сложите нескольких больших чисел, а затем сложите положительные и отрицательные числа. При манипуляциях с числами Вам поможет следующая графическая схема.

Представьте себе все целые числа в виде точек, расположенных на бесконечной линии с нулем посередине. Тогда, путешествуя вперед или назад с помощью операций сложения и вычитания, Вы можете перейти к любому числу. Например, предположим, что Вы находитесь в позиции 0 и хотите попасть в позицию -45. Очевидно, что для этого Вы должны пропутешествовать назад на 45 точек. Но добраться туда можно двумя способами: можно прибавить отрицательное число или вычесть положительное. Попробуйте проверить оба метода. В окне интерпретатора Вы должны получить результаты, показанные на рис. 3.1.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following text:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit  
(Intel)] on win32  
Type "copyright", "credits" or "license" for more info  
rmatation.  
IDLE 0.8 -- press F1 for help  
>>> 1+1  
2  
>>> 0-45  
-45  
>>> 0+-45  
-45  
>>> |
```

Рис. 3.1. $1 + 1$

Вам уже стало скучно от примитивности примеров? Давайте теперь попробуем складывать и вычитать действительно большие числа. Выполнение этой задачи с помощью карандаша и бумаги, безусловно, не вызовет у Вас никаких проблем. Даже если это будут очень-очень большие числа, просто немного терпения, и в результате обязательно получится правильный ответ. Но если Вы привыкли пользоваться современным калькулятором или раритетными в наши дни счетами, то проблема может возникнуть из-за того, что ваши числа превысят максимально допустимую разрядность Вашего электронного или деревянного инструмента.

Не сложно определить максимально допустимое число для калькулятора или счёт. Просто пересчитайте число значимых ячеек на табло или число прутиков в раме счёт. При использовании компьютера предельность вычислений не столь очевидна. Дело в том, что между внутренним представлением чисел, как его видит компьютер, и тем форматом, в котором число отображается на экране, существуют большие различия. Для внутреннего представления чисел большинство компьютеров



использует 32 бита, причём один из этих битов обычно используется для представления знака числа, указывающего, является ли данное число положительным или отрицательным. Поэтому для собственно числового значения остаётся всего только 31 бит. Таким образом, в Python самым большим допустимым целым знаковым числом является значение $2^{147} 483\,647$ (два миллиарда с мелочью).

***Прим. В. Шипкова: на конец 2005 г. в продаже по вполне доступным ценам имеются машины с 64-битами. И на самом деле 2^{32} – это не предел для Питона, расстраиваться не надо. Об этом речь пойдёт дальше.**

Конечно, Вы уже слышали термин бит прежде, но давайте ещё раз рассмотрим, что он означает, хотя бы для того, чтобы прибавить Вам уверенности. Бит – это одноразрядное значение типа "да-нет", "включено-выключено" или в числовом выражении – 0 или 1. И больше ничего иного. Чтобы представлять большие числа, необходимо объединить вместе несколько таких битов. Это как раз то, чем занимаются современные компьютеры. Они объединяют вместе свои биты в байты (8 битов) и слова (как правило, 32 бита), из которых затем и складывается весь машинный код.

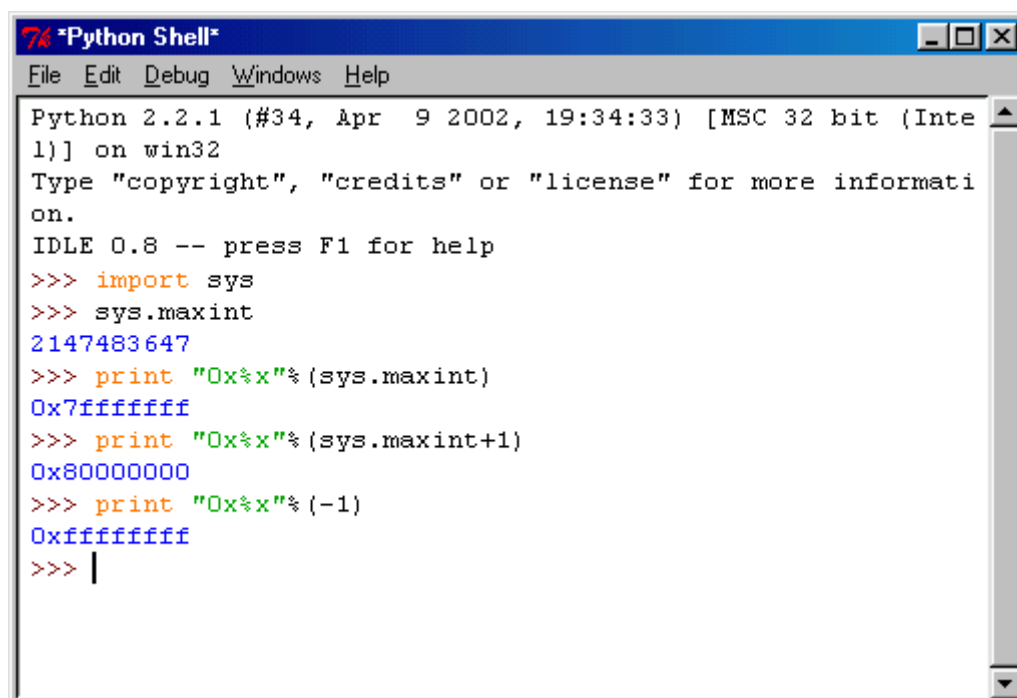
***Прим. В. Шипкова: физически значение "да-нет" кодируется наличием или отсутствием напряжения, по значению близкое к напряжению питания. Как правило 1,5...1,7 Вольт для процессора и 3,1...3,5 Вольт для оперативной памяти (сейчас могут использоваться ещё меньшие напряжения).**

На рис. 3.2 показано, что произойдет, если попытаться ввести числа, приближающиеся к этой 32-битовой границе. На показанных примерах видно, что наиболее эффективный метод использования всех 32 битов, отведённых для представления целого числа, состоит в переходе к шестнадцатеричному формату. Обусловлено это тем, что числа в шестнадцатеричном формате обрабатываются компьютером как беззнаковые, поэтому могут быть использованы все биты.

В Python есть специальное имя для самого большого знакового значения: `maxint`. Эта константа определена в специальном модуле под названием `sys`. Чтобы иметь возможность обращаться к `maxint` из своих программ, необходимо импортировать модуль `sys`. Позже Вы узнаете о том, как выполняется импортирование. Пока просто примем, что если нужен программный блок, содержащийся в каком-либо



модуле, то вместо того чтобы переписывать этот блок в своей программе, его можно импортировать из имеющегося модуля.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> import sys
>>> sys.maxint
2147483647
>>> print "0x%x"%(sys.maxint)
0x7fffffff
>>> print "0x%x"%(sys.maxint+1)
0x80000000
>>> print "0x%x"%(-1)
0xffffffff
>>> |
```

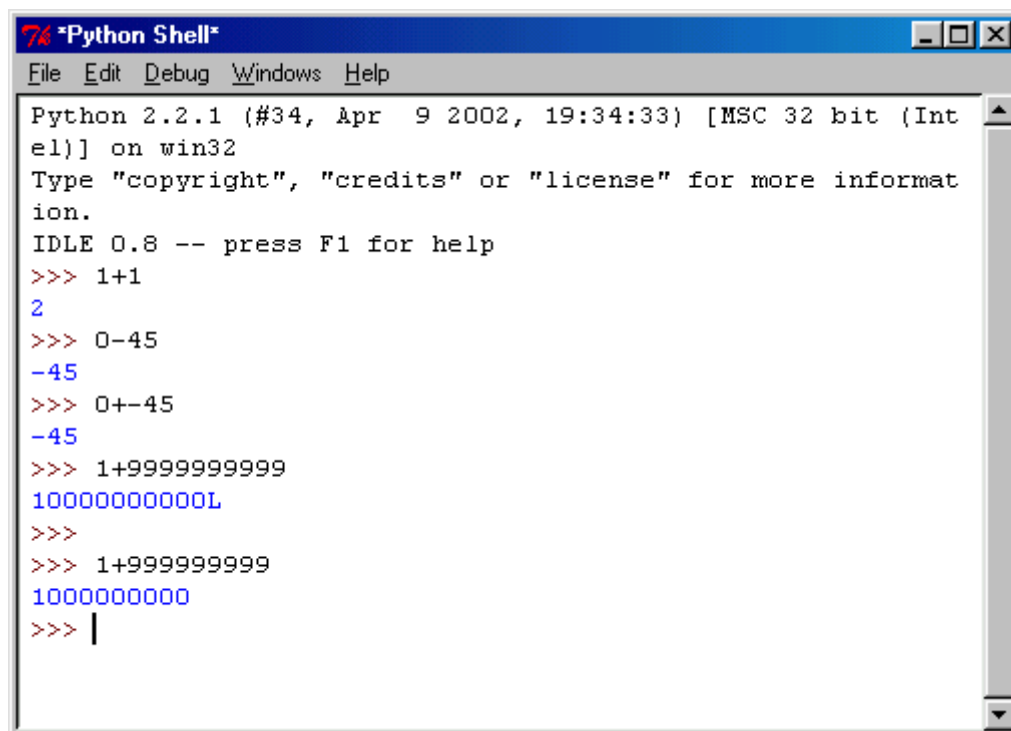
Рис. 3.2. 32-разрядные числа

Шестнадцатеричный формат является одним из способов визуализации двоичных чисел в понятном для человека виде. Двоичные числа проще преобразовывать в шестнадцатеричные, чем в десятичные. Любое 4-разрядное двоичное число можно представить одноразрядным шестнадцатеричным. Так, двоичному числу 1111₂ будет соответствовать шестнадцатеричное число F₁₆.

***Прим. В. Шипкова: все двоичные числа принято в конце отмечать буквой "b" - binary, а шестнадцатеричные буквой "h" - hex, или hexadecimal (игра слов: hex - "ведьма").**

Под термином шестнадцатеричное подразумевается представление числа по основанию 16. Это означает, что ряд шестнадцатеричных цифр следует от 0 до F, т.е. 0-15 в десятичном представлении. Если Вы хотите произвести впечатление на своих друзей и выглядеть суперкрутым компьютерным специалистом, пронумеруйте кассеты своей коллекции видеофильмов в шестнадцатеричном формате, начиная с кассеты под номером 0.

Отлично, теперь давайте посмотрим, как отреагирует Python, если в математическом действии использовать число, превышающее допустимый предел. Например, давайте прибавим к единице число 9 999 999 999. На рис. 3.3 показано, что в ответ на это скажет Python.

A screenshot of a 'Python Shell' window. The title bar says 'Python Shell' with a Python logo. The menu bar includes 'File', 'Edit', 'Debug', 'Windows', and 'Help'. The text area shows the following: 'Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32', 'Type "copyright", "credits" or "license" for more information.', 'IDLE 0.8 -- press F1 for help'. Then, several prompts and calculations are shown: '>>> 1+1' returns '2'; '>>> 0-45' returns '-45'; '>>> 0+-45' returns '-45'; '>>> 1+9999999999' returns '10000000000L' (in blue); '>>>' returns '>>>'; '>>> 1+9999999999' returns '10000000000' (in blue); and finally '>>>' followed by a cursor '|'.

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> 1+1
2
>>> 0-45
-45
>>> 0+-45
-45
>>> 1+9999999999
10000000000L
>>>
>>> 1+9999999999
10000000000
>>> |
```

Рис. 3.3. Число слишком большое, чтобы Python смог его проглотить

Мда-а... Что бы это значило? Именно то, что говорится в сообщении об ошибке: число, которое мы прибавили, оказалось слишком большим для Python, чтобы он мог правильно его обработать. Другими словами, Python попытался преобразовать число 9 999 999 999 в двоичный формат, но для этого ему не хватило битов. Мы получим аналогичный результат, если попытаемся отнять большое число. Так что же остаётся делать? Оказывается, существует очень простой метод обойти это ограничение: необходимо всего лишь сообщить Python, что он должен обрабатывать большое число именно как большое число. Для этого мы даем Python подсказку — всего лишь добавляем символ L в конце любого большого числа, если полагаем, что оно может вызвать переполнение буфера (именно это подразумевается под нехваткой битов). На рис. 3.4 показан пример такого решения.

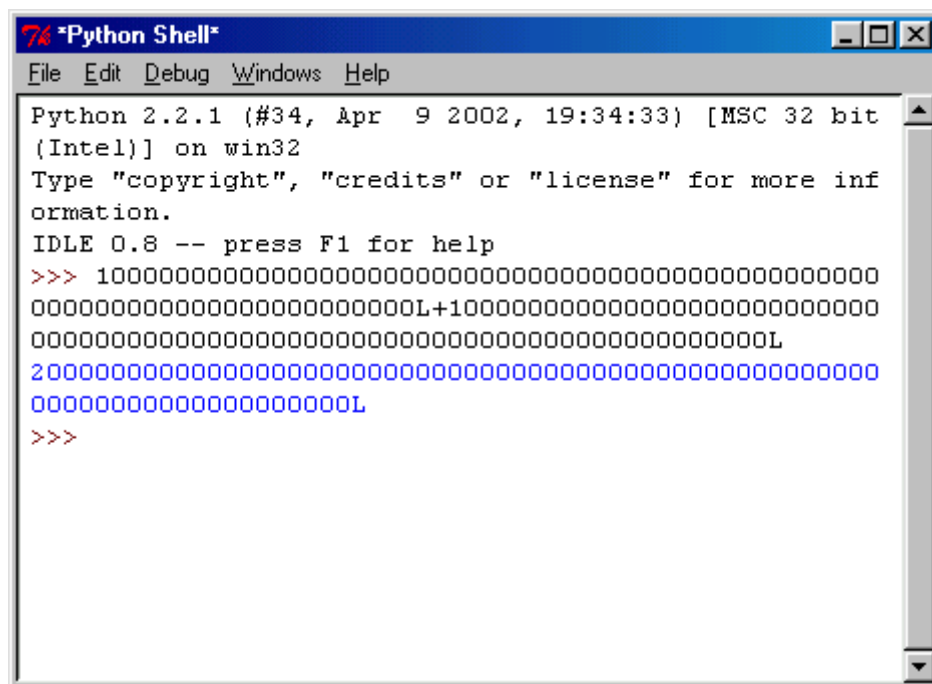


Рис. 3.5. Сложение двух длинных чисел

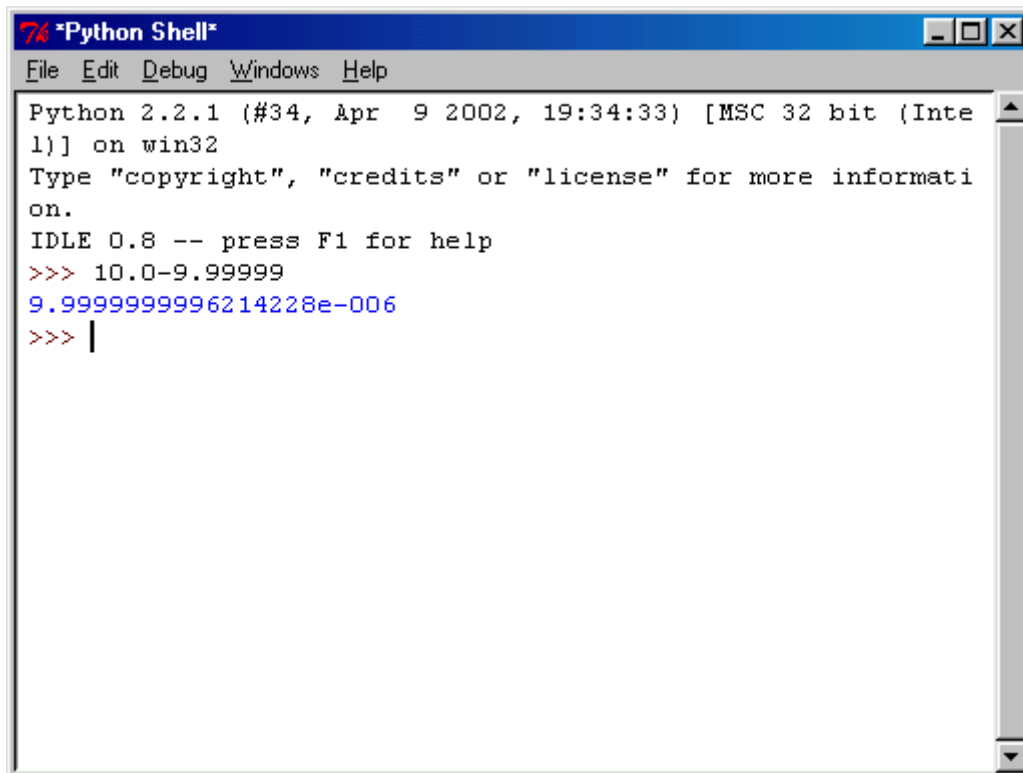
Число, состоящее из единицы с сотней нулей, называется *гугол* (googol). Это название было введено в обиход американским математиком Эдвардом Каснфом (Edward Kasner), а придумано его девятилетним племянником, Милтоном Сироттой (Milton Sirota).

Вы можете складывать обычные целые числа и длинные целые числа без каких бы то ни было проблем. Правилами Python устанавливается, что всякий раз, когда Вы выполняете арифметические функции с любыми двумя операндами, меньший операнд приводится к типу большего. Таким образом, если Вы суммируете `1 + 1000000000000000000000000000L`, Python автоматически преобразует `1` в `1L`.

Математические операторы, например суммирования (+) и вычитания (-), оперируют с операндами. Например, выражение "1 + 4" обозначает: "применить оператор суммирования к операндам 1 и 4".

Вам придётся также столкнуться ещё с третьим типом чисел — числа с плавающей запятой. Это числа, в которых есть десятичная запятая, например 1,1 и 3,14159265359. (В англоязычных странах для представления десятичных чисел вместо плавающей запятой используется точка (!!!). Обратите на это внимание и не удивляйтесь, что когда в тексте мы говорим о плавающей запятой, то в примерах ставим точку. Примечание: использование запятой в коде программы Python вызовет ошибку.) Иногда в результате сложения и вычитания таких чисел выводится результат, который Вы даже и не предполагали получить. Так, к примеру, на рис. 3.6 показано

выражение, результатом которого Вы, скорее всего, ожидали бы увидеть 0,00001.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> 10.0-9.99999
9.9999999996214228e-006
>>> |
```

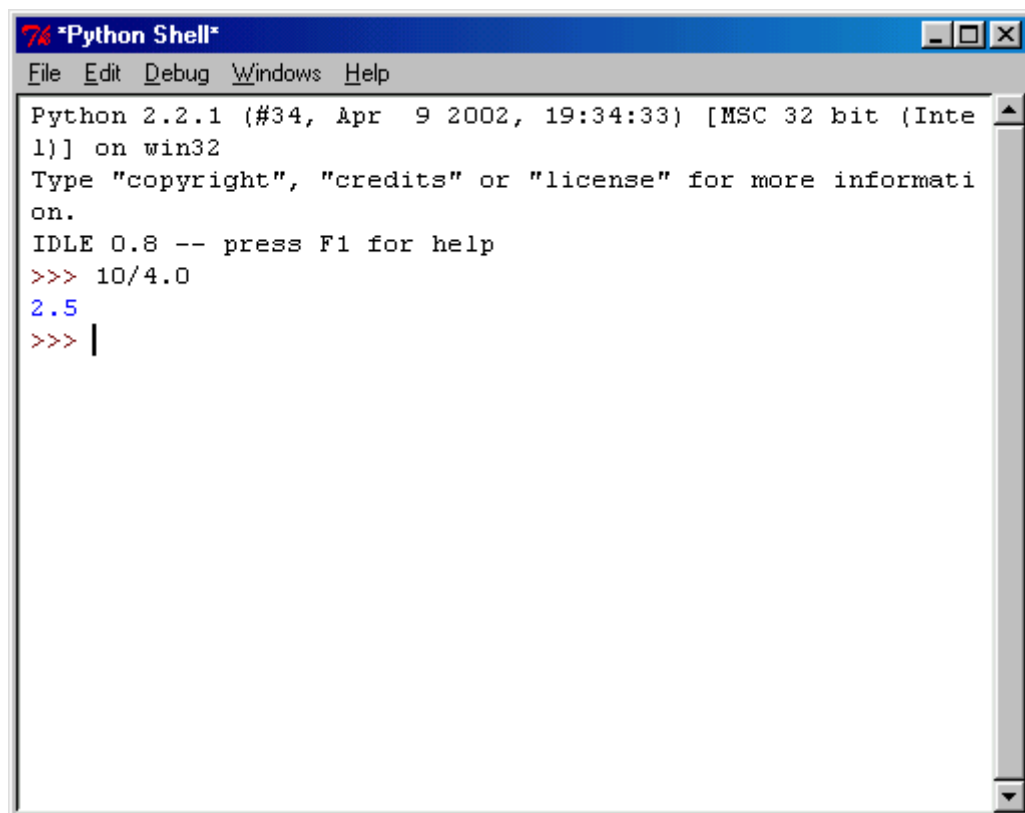
Рис. 3.6. Вычитание чисел с плавающей запятой

В общем-то, Вы такой результат и получили. Не удивляйтесь, просто число представлено в экспоненциальном формате, о котором мы поговорим более подробно в последнем разделе этой главы. А пока перейдем к следующей теме.

Умножение, отношение и деление по модулю

В предыдущем разделе Вы узнали о сложении и вычитании обычных целых чисел, длинных целых чисел и чисел с плавающей запятой. В данном разделе Вы познакомитесь с умножением, делением и с кое-чем таким, о чем Вы могли ещё не слышать, — делением по модулю. При этом будут использоваться те же типы чисел.

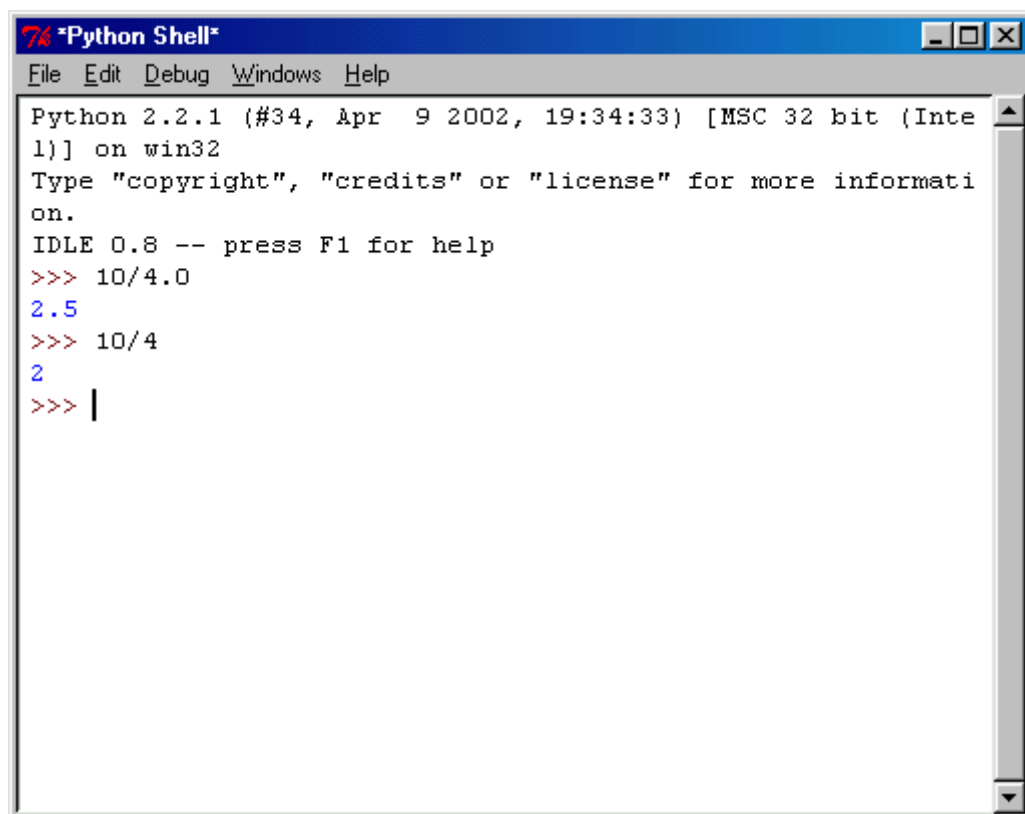
И снова позволю себе повториться — правила выполнения этих операций в Python не очень отличаются от тех, которые Вам знакомы со школы. Единственный нюанс, который необходимо всегда учитывать, состоит в том, что тип операнда в значительной степени определяет то, как Python будет обрабатывать числа. Например, деление числа 10 на 4,0, как показано на рис. 3.7, даёт ожидаемый результат — 2,5.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following text: "Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32", "Type 'copyright', 'credits' or 'license' for more information.", "IDLE 0.8 -- press F1 for help", and a prompt ">>>". The user has entered "10/4.0" and the shell has responded with "2.5". The prompt ">>>" is followed by a vertical bar cursor.

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> 10/4.0
2.5
>>> |
```

Рис. 3.7. Деление чисел

Но в случае, показанном на рис. 3.8, мы получаем совершенно иной ответ.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following text: "Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32", "Type 'copyright', 'credits' or 'license' for more information.", "IDLE 0.8 -- press F1 for help", and a prompt ">>>". The user has entered "10/4.0" and the shell has responded with "2.5". The user has then entered "10/4" and the shell has responded with "2". The prompt ">>>" is followed by a vertical bar cursor.

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> 10/4.0
2.5
>>> 10/4
2
>>> |
```

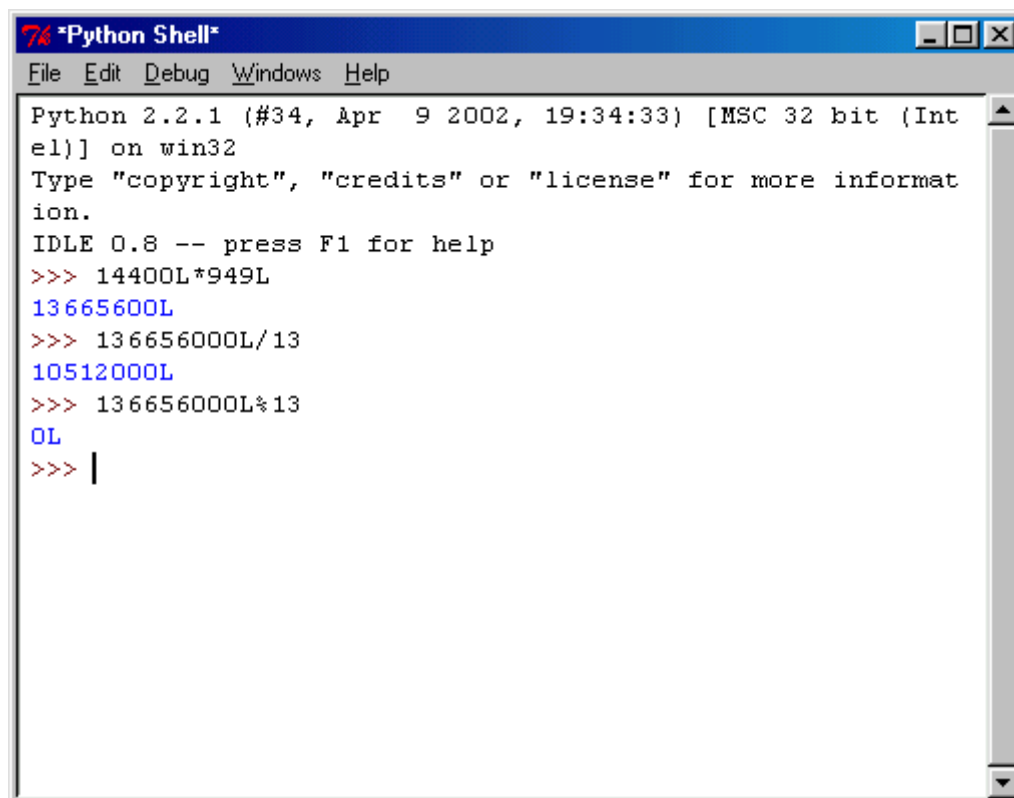
Рис. 3.8. ещё один пример деления

Несовпадение ответов объясняется тем, что в первом случае, указав десятичную запятую (точку) в операнде 4.0, мы сообщили Python, что нам небезразлична информация об остатке деления после десятичной запятой. Во втором случае, опустив десятичную запятую, мы явно указали Python, что нас интересует только целая часть числа. И Python любезно отбросил дробную часть числа (.5), о которой мы сообщили ему, что в ней не нуждаемся. Таким образом, как и в случае с операциями сложения и вычитания, Python определяет формат результата по форматам переданных ему операндов. Поэтому Вам следует внимательно относиться к типам операндов. Очень просто потерять десятичную запятую там, где в действительности Вы ожидаете её увидеть. Такая потеря в дальнейшем может привести к неправильной работе программы.

Та лёгкость, с которой при делении можно получить неожиданный или даже неправильный результат, была и остаётся темой жарких дебатов, ведущихся и по сей день среди участников телеконференций по тематике программирования на Python. Некоторые респонденты доказывают необходимость применения специального оператора, отличного от обычного оператора деления (/), который бы указывал Python, что требуется выполнить целочисленное деление (как показано на рис. 3.8), или наоборот, устанавливающего для результата формат числа с плавающей запятой (рис. 3.7). Так, в качестве специальных операторов деления предлагалось использовать сочетание символов /. для деления только с плавающей запятой и сочетание // – только для целочисленного деления. Но эти предложения не нашли поддержки у Гуйдо, хотя он обещал подумать над этим вопросом и внести некоторые изменения в будущие версии Python.

***Прим. В. Шипкова: я не поленился проверить верно ли для версии 2.4.1 деление с "бек слешем" – обратной косой чертой. Но увы, воз и ныне там. А надо бы.**

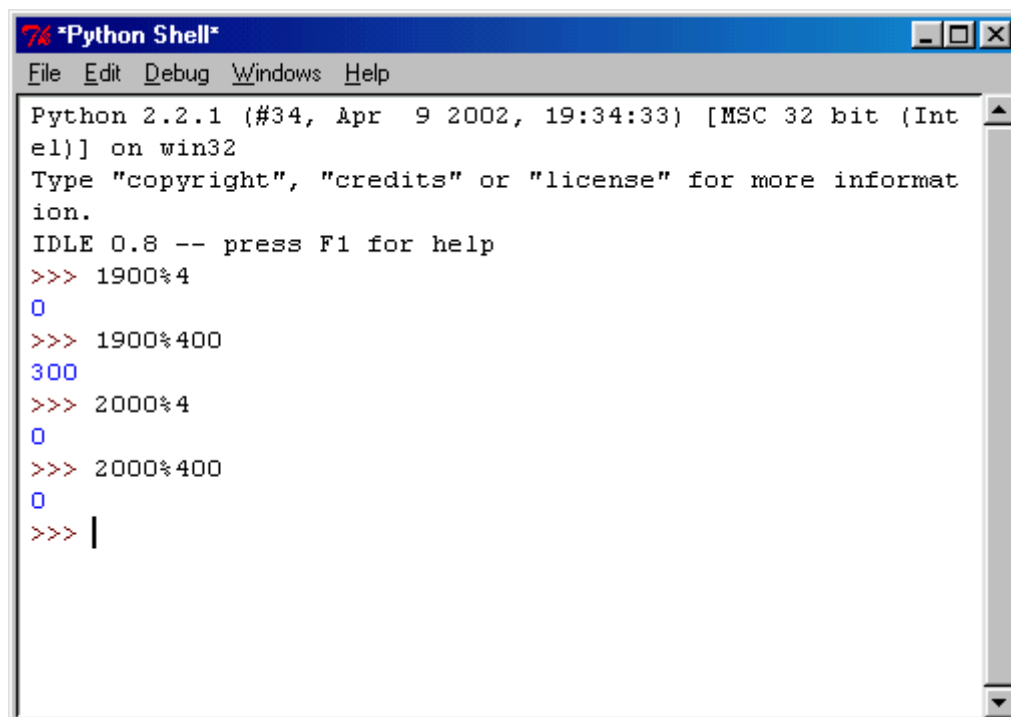
Как и в случаях со сложением и с вычитанием, длинные целые числа можно использовать в выражениях умножения и деления (рис. 3.9).



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> 14400L*949L
13665600L
>>> 136656000L/13
10512000L
>>> 136656000L%13
0L
>>> |
```

Рис. 3.9. Деление и умножение длинных чисел

Обратите внимание, что в третьем примере, показанном на рис. 3.9, используется новый оператор, обозначенный символом `%` — деление по модулю. С помощью этого оператора можно осуществлять деление обычных или длинных целых чисел, причём от полученного результата отбрасывается целая часть числа (частное), а остаток возвращается. Например, выражение `136656000L % 13` возвратит нуль, потому что число 136 656 000 делится на 13 без остатка. Деление по модулю — очень удобное средство, например при вычислении календарных дат. Вот один из примеров. Практически во всех календарях присутствует високосный год, который можно вычислить делением по модулю. А в тех календарях, где високосные годы отсутствуют, ситуация настолько сложна, что без деления по модулю вообще не обойтись. Примером последнего может быть календарь Майя. С нашим григорианским календарем ситуация попроще, так как високосные годы в нём присутствуют. В юлианском календаре, от которого собственно и произошел григорианский, действует очень простое правило високосного года: любой год, значение которого делится без остатка на 4, является високосным. Правило григорианского календаря вносит поправку в эту формулировку, утверждая, что годы столетий (1700, 1900 и т.д.) являются високосными только в том случае, если они делятся без остатка на 400. На рис. 3.10 показано, как определить, является ли 2000-й год високосным.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> 1900%4
0
>>> 1900%400
300
>>> 2000%4
0
>>> 2000%400
0
>>> |
```

Рис. 3.10. Определение високосного года

Год 2000 — високосный, а 1900 — нет. Я как-то читал один детектив, в котором следователь разбил "железное" алиби подозреваемого после того, как сообразил, что 1900-й год не был високосным. Я уже не помню в деталях ни саму историю, ни её автора, но факт остаётся фактом — иногда бывает важно быстро определить, является ли указанный год високосным.

Существует специальная функция, которую можно использовать для возвращения частного и остатка от деления. Функция `divmod(x,y)` возвращает результаты выполнения двух действий: x/y и $x\%y$. Вы можете проверить это, введя в окне IDLE команду `print divmod(53,13)`. Пара результатов будет показана в виде `(4,1)`. В Python всегда возвращаются группы результатов, о чем мы поговорим позже.

Округление, функции `floor()` и `ceil()`

В этом разделе мы будем иметь дело исключительно с числами в формате с плавающей запятой. Во всех случаях, когда необходимо выполнить любую математическую операцию над числами с плавающей запятой, следует помнить об ограничениях, связанных с базовым представлением этих чисел компьютером. Компьютеры оперируют с числами только в двоичном формате, т.е. с основанием 2. В числах с основанием 2 присутствуют только две цифры — 0 и 1. Это идеальная система счисления для компьютеров, поскольку значения 0 и 1 могут быть представлены простым двух-позиционным переключателем. Электролампочка, тумблер, электросхемы на лампах или транзисторах — все это примеры

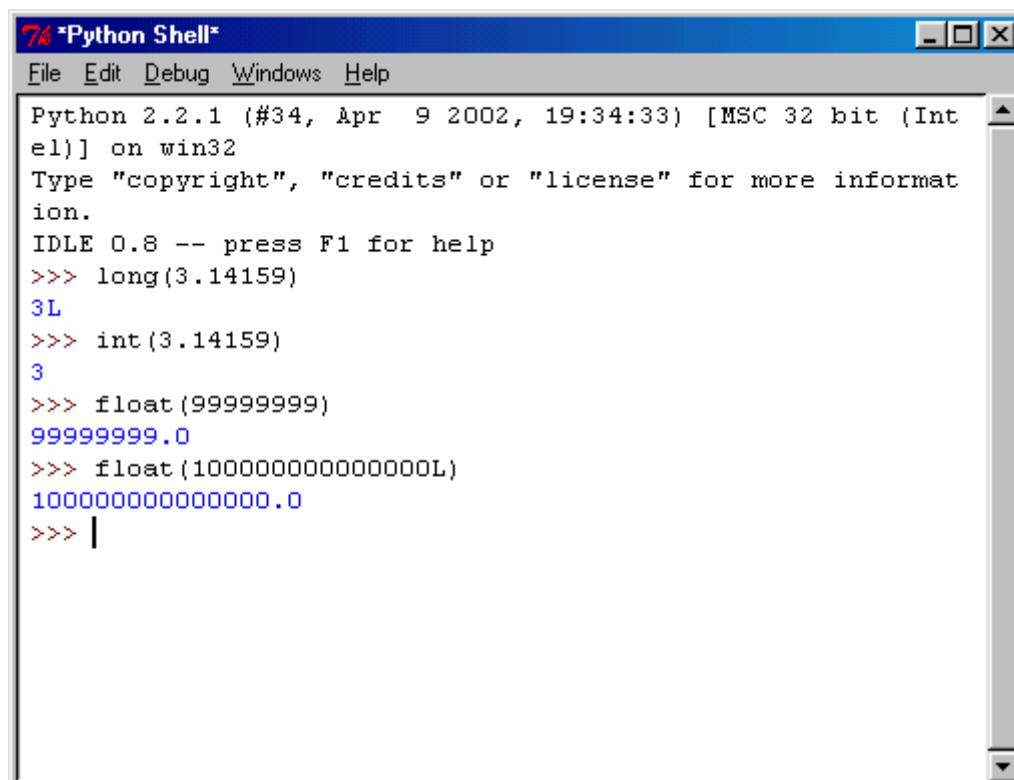
двухпозиционных переключателей. Преобразование целых чисел, представленных на базе одной основы, к формату с другой основой является относительно простой процедурой, которой занимаются компьютеры и неплохо справляются с момента их появления. При смене основания числа не теряется ни "капли" информации. Всегда можно преобразовать числа к другой основе без какого-либо округления. Например, десятичному числу 15 точно соответствует шестнадцатеричное число F, и любое другое десятичное число однозначно соответствует определённому шестнадцатеричному, восьмеричному, двоичному и т.д. В таком случае говорят, что при преобразовании целых чисел *потери точности* не происходит.

Но это утверждение справедливо только для целых чисел. Все меняется, стоит только вставить в число десятичную запятую. Способы, с помощью которых в компьютерах осуществлялось представление чисел с плавающей запятой, исторически были разнообразными, сложными и довольно противоречивыми. Только в течение последних нескольких лет появилось что-то похожее на стандарт представления чисел этого типа. Тем не менее многие полагают, что даже этот стандарт, IEEE-64, полон ловушек и недостатков. К сожалению, чтобы полностью раскрыть эту увлекательную тему, понадобится написать отдельную книгу не меньшего объема, чем данная. Но вряд ли такая книга выйдет, так как найдется не много людей, которые проявили бы к ней интерес. Уж больно это специфический вопрос для узкого круга специалистов. Пока вполне достаточно будет сообщить Вам, что проблем в этом вопросе множество, а их решение сопряжено со значительными трудностями.

На данном этапе Вам достаточно знать, что при работе с числами с плавающей запятой возникают проблемы, связанные с потерей точности. Причем довольно неожиданным оказывается тот факт, что умножение и деление при таких обстоятельствах сопровождается определённой погрешностью, которая умеренно возрастает при повторных операциях, тогда как повторное сложение и вычитание существенно увеличивают погрешность. Из этого следует мораль — необходимо с осторожностью и умеренностью использовать сложение и вычитание чисел с плавающей запятой, а из двух способов решения задачи выбирать тот, где умножение и деление превалируют над суммированием и вычитанием.

Когда появляется необходимость выполнять математические действия с числами с плавающей запятой, рано или поздно возникает потребность преобразовать некоторые результаты, представленные в формате числа с плавающей запятой, в целые числа или в длинные целые числа. Для выполнения этой задачи

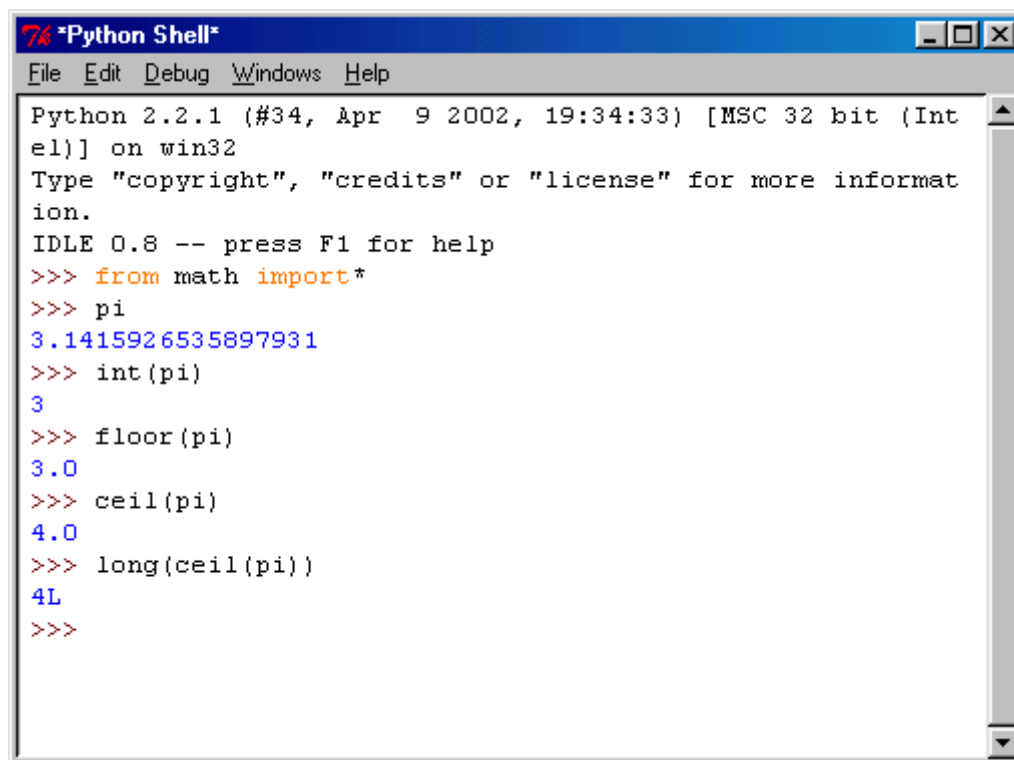
в Python имеется простой способ, который состоит в том, чтобы использовать встроенные функции преобразования типов. Несколько примеров их применения показаны на рис. 3.11.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following text:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> long(3.14159)
3L
>>> int(3.14159)
3
>>> float(999999999)
999999999.0
>>> float(1000000000000000000L)
1000000000000000000.0
>>> |
```

Рис. 3.11. Несколько примеров преобразования типов

Преобразования типов часто сопровождаются округлением значения. Иногда требуется округлить результат к большему значению, иногда — к меньшему. Для этого существуют специальные методы округления, позволяющие контролировать данную операцию: `floor()` и `ceil()`. Помните, как раньше мы уже импортировали модуль `sys`? Точно так же всякий раз, когда Вам понадобится использовать математические функции, следует импортировать модуль `math`. Но теперь мы воспользуемся другой инструкцией импортирования модулей (рис. 3.12).

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following text:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> from math import *
>>> pi
3.1415926535897931
>>> int(pi)
3
>>> floor(pi)
3.0
>>> ceil(pi)
4.0
>>> long(ceil(pi))
4L
>>>
```

Рис. 3.12. Функции *floor()* и *ceil()*

Данная инструкция импортирования позволяет обращаться к членам модуля непосредственно, без указания имени модуля. Если бы мы импортировали модуль `math` иначе, то чтобы вызвать число π (пи), нам пришлось бы вводить `math.pi`. Точно так же для обращения к функциям `floor()` и `ceil()` пришлось бы использовать вызовы `math.floor()` и `math.ceil()`. Многие люди предпочитают не использовать инструкцию `from x import *`, так как это небезопасно. Дело в том, что имена в импортированном модуле могут конфликтовать с теми, которые программист определил в своей программе. Впрочем, вероятность появления таких конфликтов невелика. Кроме того, ничего страшного не произойдет, просто интерпретатор сообщит Вам об ошибке. С другой стороны, профессиональный программист никогда не станет импортировать модуль с помощью `import *`, если он недостаточно с ним знаком. Даже если ничего плохого не произойдет, это дурной тон программирования. Прежде чем что-либо импортировать, следует достать документацию на этот модуль и хорошо с ней познакомиться.

На рис. 3.12 показано практически все, что необходимо знать для правильного использования функций округления и преобразования типов. Только проверьте, чтобы количество открывающих круглых скобок соответствовало количеству закрывающих. И ещё на один момент хотелось бы обратить Ваше внимание. Посмотрите, как функция `long()` принимает в качестве аргумента другую функцию `ceil()`. Это важный момент, который необходимо хорошо запомнить: вместо чисел в

качестве аргументов функций допускается использовать другие функции, которые обрабатывают свои аргументы.

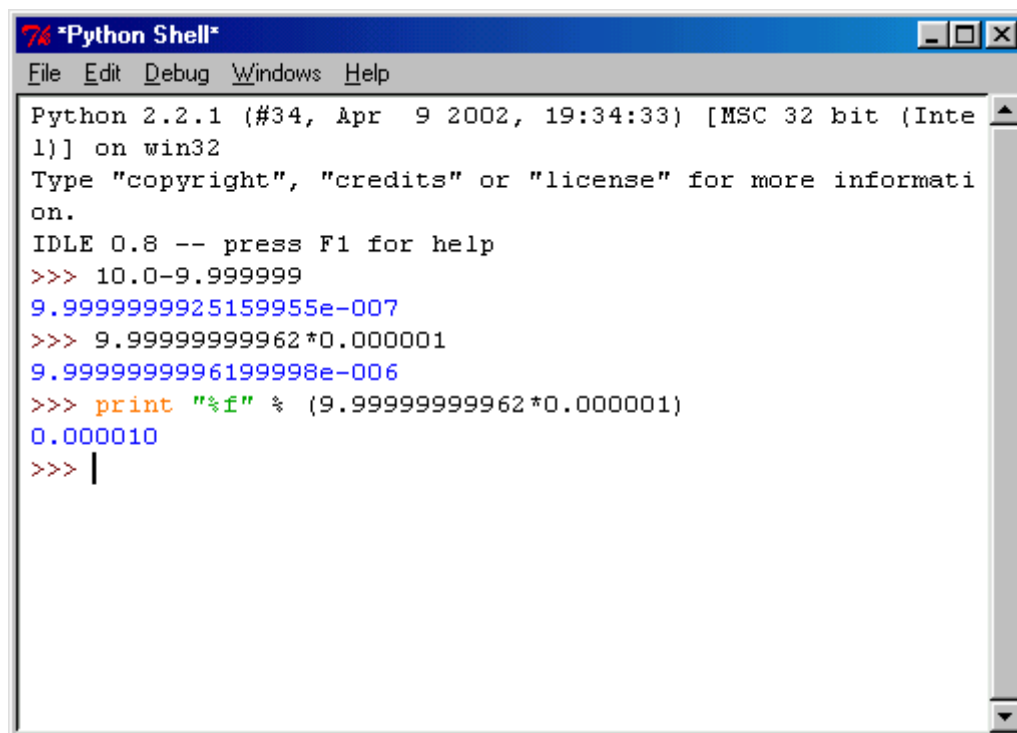
Может возникнуть ещё один вопрос, связанный с округлением чисел с плавающей запятой, — как будут округляться отрицательные числа? Мы ненадолго отложим обсуждение этого вопроса и вернёмся к нему позже в этой главе.

Возведение в степень

Возведение в степень, как Вы должны помнить со школы, это, попросту говоря, умножение некоторого числа на себя определенное количество раз. Например, 2^2 означает умножить два на два, или возвести два в квадрат, что в результате даст 4. Точно так же 2^3 означает, что надо умножить два на два и на два, или возвести двойку в куб, что в результате даст 8.

***Прим. В. Шипкова: 2^2 или 2^3 – так вообще в программировании принято записывать степенные выражения. В Python принята несколько иная запись – $2**2=4$, $2**3=9$.**

Экспоненциальное представление чисел, или, выражаясь более формально, форма записи чисел с плавающей запятой, использует показатель степени, чтобы при работе с очень большими числами избежать записи с многократным повторением нулей. Ранее, на рис. 3.6, был показан пример, в котором мы вычитали $10.0 - 9.99999$ и получили неожиданный ответ: $9.99999999962e-006$, тогда как предполагали увидеть 0.00001 . Экспоненциальная форма записи содержит две части: показатель степени и дробную часть (которую иногда неправильно называют мантиссой). Что касается нашего удивительного ответа, то $e-006$ как раз и является показателем степени, а 9.99999999962 — её дробная часть. Символ e как раз и указывает начало показателя степени (от английского *exponent*). Значение -006 — это фактическое значение показателя степени. Число, записанное подобным образом, обычно произносится как "9.99999999962 на 10 в минус шестой". Все очень просто, затруднение может состоять только в том, чтобы выговорить все эти девятки после десятичной запятой. Отрицательное значение показателя степени сообщает нам, что необходимо разделить единицу на значение нашего основания, т.е. на 10, определенное количество раз. К примеру, 10^{-1} является тем же самым, что и дробь $1/10$ или $0,1$. Показатель степени -6 в нашем примере означает разделить единицу на 10 шесть раз, или $1/1\ 000\ 000$. Этой дроби соответствует число 0.000001 . Чтобы убедиться в этом, посмотрите на рис. 3.13.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The text area shows the following content:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
IDLE 0.8 -- press F1 for help
>>> 10.0-9.9999999
9.9999999925159955e-007
>>> 9.99999999962*0.000001
9.9999999996199998e-006
>>> print "%f" % (9.99999999962*0.000001)
0.000010
>>> |
```

Рис. 3.13. Отрицательные показатели степени

Обратите внимание на последний результат, выведенный на экране, — число 0.000010. Это значение чрезвычайно близкое к тому, что мы ожидали увидеть с самого начала. Помните, что я говорил раньше о погрешностях округления при сложении и вычитании, которые имеют тенденцию существенно возрастать? Когда Вы оперируете с очень маленькими числами, порядка 10^{-6} , эти ошибки суммируются и проявляют себя в значительной степени.

Экспоненциальное представление чисел весьма широко используется в программировании и научно-исследовательских работах. Поэтому имеет смысл познакомиться с этой формой записи поближе. Чаще всего Вам могли встречаться следующие значения: 10^3 , т.е. 1000, 10^6 , или миллион, 10^9 — миллиард и 10^{12} — триллион. (Интересно, что термины, применяемые к числам свыше миллиона, в разных странах могут иметь различные значения. Так, согласно американской терминологии, слово биллион (billion) обозначает миллиард.) А помните единицу, за которой следовала сотня нулей — гугол? Мы использовали это число в примерах несколько раньше. Так вот, для записи гугола в экспоненциальном представлении существуют два способа. Первый — в виде 10^{100} , а второй — в виде 10^{10} (?-что то не правильно указано). Но в обоих случаях мы имеем одно и то же число. После того как Каснер ввёл в обиход термин гугол, другой математик предложил число под названием гуголплекс (googolplex), которое соответствует $10^{\text{гугол}}$. Боюсь, нам не хватит бумаги, чтобы записать это число в обычном виде. Информация для

любопытных: на Web-странице этой книги можно найти программу на языке Python, которая выводит на печать числа гугол и гуголплекс. Она будет выполняться до тех пор, пока у Вас не лопнет терпение, или не исчерпается компьютерная память, или свободное пространство на жёстком диске компьютера, или пока Земля не упадет на Солнце (неизвестно, что произойдет первым). На рис. 3.14 показан быстрый метод вывода числа гугол в Python.

[illegible]

Рис. 3.14. Вывод числа гугол

Применение скобок

Разобравшись с разными типами данных и форматами вывода числовых значений, перейдем на следующую ступеньку лестницы – написание математических формул. Безусловно, с математическими формулами Вы встречались не раз. Практически во всех этих формулах, за исключением самых элементарных, встречаются круглые скобки. Не правда ли, Вы это заметили? Круглые скобки присутствуют в формулах для того, чтобы указать порядок выполнения действий. Другими словами, взглянув на такую формулу, сразу видно, с чего именно необходимо начинать для её выполнения, даже если математическое действие в скобках Вам совершенно не знакомо и выглядит полной абракадаброй.

Python, как и все другие языки программирования, имеет сложную систему правил, называемых правилами приоритета операторов. С учётом этих правил опытный программист может управлять порядком выполнения математических формул без

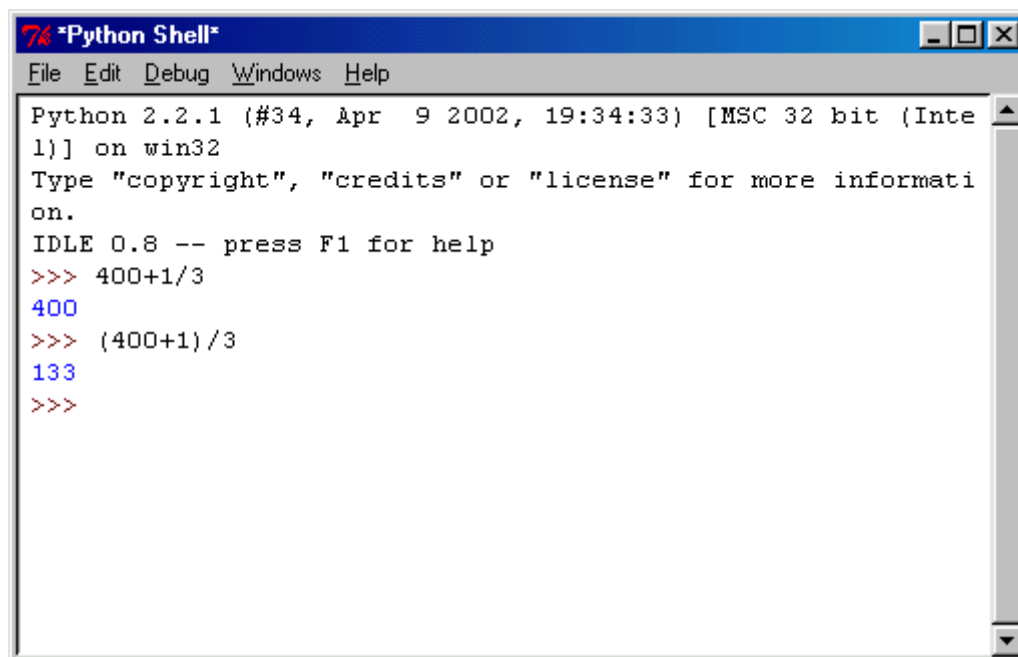
множества скобок. Это может повысить читабельность формул, если, конечно, соблюсти меру и следовать логике. Можно было бы и вообще обойтись без скобок. Для этого всего лишь необходимо запомнить каких-то 20–30 сложных правил, переписать свои математические выражения таким образом, чтобы они удовлетворяли этим правилам, но не соответствовали никакой человеческой логике, и тогда в Вашем коде не разберётся ни один шпион. Когда будете писать свои формулы, найдите золотую середину.

Как Вы поняли, лично я не большой поклонник следования правилам приоритета операторов. Я люблю, чтобы всё было, разложено по полочкам, причём как можно проще и явно. К тому же, я предпочитаю указывать компьютеру, что он должен для меня сделать, а не следовать его указаниям. Обобщая свой опыт работы, я установил свои собственные правила, которые, как мне кажется, намного проще.

Правило 1 Если в формуле используются только действия сложения или вычитания, забудьте вообще о круглых скобках.

Правило 2 Если операции сложения-вычитания перемежаются с умножением-делением, всегда используйте скобки.

Когда я делал только первые шаги в программировании, я несколько раз обжигался на правилах приоритета выполнения операторов, так как они не всегда следуют общепринятой логике. Скобки в этом плане более надёжны, так как обладают абсолютным приоритетом. Рассмотрим применение скобок на простом примере, показанном на рис. 3.15. Если нам нужно найти отношение суммы на число, то следующее выражение будет неправильным: $400 + 1 / 3$. Чтобы исправить ситуацию, заключите сумму $400 + 1$ в скобки.



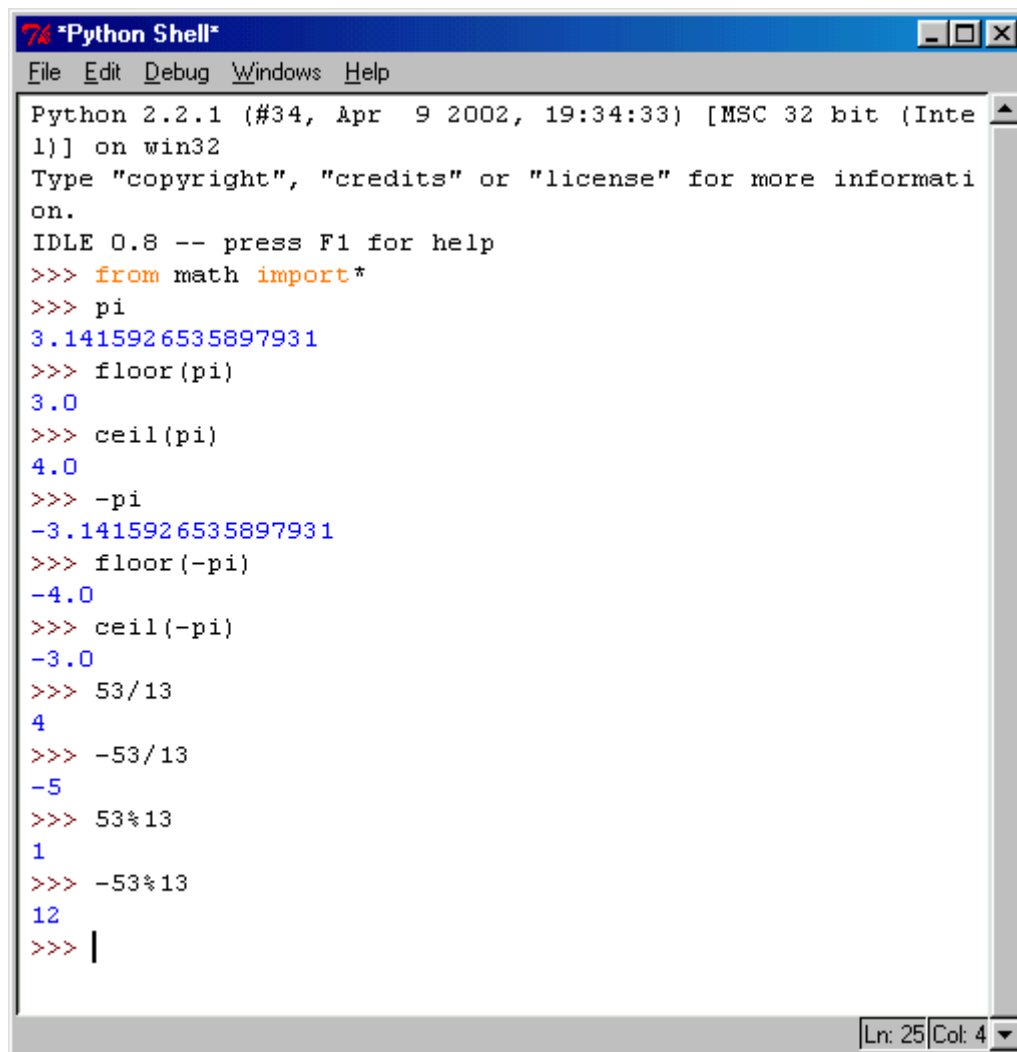
```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
IDLE 0.8 -- press F1 for help
>>> 400+1/3
400
>>> (400+1)/3
133
>>>
```

Рис. 3.15. Пример использования скобок

Если ваша формула слишком длинная, то хоть используйте скобки, хоть не используйте, читабельной она никогда не станет. Попробуйте разбить формулу на несколько строк. Это улучшит её читабельность, что Вы почувствуете во время отладки программы. Помните также о том, что когда-нибудь с вашим кодом может работать кто-то другой. Не доводите Вашего коллегу до психоза, тем более что вернувшись к своей собственной программе через пару месяцев, Вы сами окажетесь не в лучшем положении. Поэтому критическим взглядом пройдитеесь по каждой строке кода и спросите себя: "Смогу ли я вспомнить через шесть недель, что выполняет данная строка?" Если возникают сомнения, то формулу лучше переписать прямо сейчас.

Некоторые нюансы и секреты

Одной из особенностей Python, которую необходимо учитывать при обработке чисел любого типа, является механизм выполнения функций `floor()` и `ceil()` для отрицательных аргументов. По выполнению этих функций Python отличается от языков C и C++, поэтому если Вы работали с любым из этих языков, то должны быть начеку, чтобы предупредить появление ошибок. Данная проблема проиллюстрирована на рис. 3.16.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following text:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> from math import *
>>> pi
3.1415926535897931
>>> floor(pi)
3.0
>>> ceil(pi)
4.0
>>> -pi
-3.1415926535897931
>>> floor(-pi)
-4.0
>>> ceil(-pi)
-3.0
>>> 53/13
4
>>> -53/13
-5
>>> 53%13
1
>>> -53%13
12
>>> |
```

The status bar at the bottom right shows "Ln: 25 Col: 4".

Рис. 3.16. Округление отрицательных чисел

В языке C применение функции `floor()` к аргументу `pi` даёт результат 3, но если применить её к значению `-pi`, то получим `-3`, а в Python — `-4`. Если же используется функция `ceil()`, то получится `-4` и `-3` соответственно. Почему так происходит? По определению, в обоих языках функция `floor()` осуществляет "округление вниз". Почему же тогда получаются разные результаты? Дело в том, что эти языки по-разному понимают, что такое "округление вниз". В случае с Python слово вниз подразумевает меньше, аналогично тому, как $0,0001 < 0 < 10000$ и число 0.00000000000001 меньше числа 0.0001 . Тогда как в C меньше означает ближе к нулю. Таким образом, в Python число `-4` "лежит ниже", чем `-3.14159265359`. При целочисленном делении результат получается таким же, как при применении функции `floor()`. Результат округляется к меньшему числу, а не к тому, которое находится ближе к нулю.

Эти несостыковки в логике языков могут привести к неожиданным проблемам. Например, работая в C, я написал функцию, которая вычисляла разницу в днях между

григорианским и юлианским календарями. Эта разница носит название отклонение и является константой в промежутке от 100 до 200 лет. Я перенес эту функцию в Python и быстренько проверил её работоспособность. Казалось, что все работает нормально. Но через два-три дня после этого мне подвернулся случай опробовать её на отрицательных датах. Результаты её работы утверждали, что до 0-го года не было никаких отклонений. (В этой программе годы до нашей эры выражались отрицательными цифрами.) Это была явная ошибка. Я исследовал код функции и вышел на строку, которая содержала следующее выражение: $devn = 2 - leaper + -(leaper/4)$. В этом выражении `devn` и `leaper` являются переменными, назначение которых состоит в сохранении значений. (С переменными Вы познакомитесь в следующей главе.) Переменная `leaper` содержит значение столетия. Если это значение выражалось отрицательной величиной, то результат отношения `leaper/4` округлялся до меньшего значения, а не до того, которое ближе к нулю. Поэтому программа работала в C, но допускала ошибки в Python.

В целом вычисления с датами в Python выполнять даже удобнее, чем в других языках. Новичкам будет проще. Они воспримут это как должное, так как на них не давит груз знаний о том, как работают аналогичные функции в других языках. Я же знал, как должна была выполняться эта функция, и был просто обескуражен, когда она не работала ожидаемым образом. Моя ошибка состояла в том, что я перенес код с языка C на Python, не проверив должным образом, насколько их функции соответствуют друг другу.

Если у Вас богатый опыт программирования, заставьте себя внимательно изучить средства Python, вчитываясь во все детали, чтобы не допускать подобные оплошности.

Полезное свойство Python, которое лично мне очень понравилось, — встроенная поддержка длинных целых чисел. Я был разочарован, работая в C и C++ над календарем Майя. Эти языки строго отслеживают тип данных, и чтобы иметь возможность обрабатывать в программе как обычные, так длинные целые числа, приходилось дважды переписывать код для обоих типов. Для Python достаточно явно указать длинное число и для обработки его использовать те же функции, что и для обычного числа. Конечно, Python стал бы ещё удобнее, если бы вообще не видел разницы между обычными и длинными числами. Действительно, различие между этими типами чисел совершенно условно и связано только с объёмом памяти, предоставляемым для сохранения значения. Guido обещал в скором будущем сделать прозрачным переход от типа к типу. А пока Вам придётся проявлять внимательность с

арифметическими операциями, при выполнении которых возможно переполнение целых 32-битовых знаковых чисел. Если Вам, к примеру, случится столкнуться с календарем народов Майя, то Вы не успеете и глазом моргнуть, как преодолеете двухмиллиардный (плюс мелочь) предел.

Существуют встроенные методы автоматического преобразования всех целых чисел в программе в длинные целые, но мы не будем сейчас останавливаться на них. Можно также вручную пометать все значения в программе как длинные. Но за подобное упрощение придётся расплачиваться понижением быстродействия. Впрочем, для большинства программ прекрасно подходят обычные 32-разрядные целые числа.

Резюме

В этой главе мы рассмотрели все основные арифметические операторы языка Python и опробовали их с помощью интерпретатора. Вы должны были ощутить атмосферу работы с Python и привыкнуть к окну IDLE. Надеюсь, что в следующей главе, когда мы начнём обсуждать переменные и вопросы управления потоками данных, Вы будете довольно комфортно чувствовать себя. Вы также узнали, как выполняется округление чисел, каковы особенности обработки чисел с плавающей запятой, что такое экспоненциальное представление чисел, зачем применяются скобки. Кроме того, теперь Вы знаете, что логика Python несколько отличается от логики других языков программирования.

Для обобщения материала посмотрите табл. 3.1, в которой представлены все математические операторы, используемые в Python. Некоторые из этих операторов будут подразумевать совершенно иное действие, когда будут применяться к объектам, не являющимся числами. Мы будем подробно рассматривать особенности применения этих операторов по мере освоения нового материала.

Таблица 3.1. Математические операторы в Python

	Символ	Значение
1	+	Сложение/тождество
2	-	Вычитание/отрицание
3	*	Умножение
4	/	Деление
5	%	Деление по модулю

3. Если двоичное число по основанию 2,
а шестнадцатеричное по основанию 16, то что
такое восьмеричное число и где оно используется?
- а) Числа с основанием 32; используются программистами Macintosh.
б) Это понятие не имеет никакого отношения к числам; термин *восьмеричное (octal)* – это название символа #.
в) Числа с основанием 8; используются программистами UNIX.
г) Числа с основанием 8; но никто их не использует.

Ответы

1. Можно использовать все указанные методы.
2. г. Для чисел с плавающей запятой в Windows и Linux максимальным установлено значение с показателем степени ± 308 . Это достаточно большое (или достаточно маленькое) число. При выходе за эти пределы библиотеки, поддерживающие вычисления с плавающей запятой, возвращают стандартный символ бесконечности.
3. в. Восьмеричное число – конечно же, это число с основанием 8. И числа в таком формате действительно главным образом используют программисты, разрабатывающие приложения для UNIX. Наиболее наглядным примером применения таких чисел является программа *chmod* для UNIX, которая позволяет изменять права доступа к файлам. А символ #, действительно, имеет необычное название – *октоторп (octothorpe)*.

Примеры и задания

Если Вас беспокоит проблема потери точности вычислений с использованием чисел с плавающей запятой, познакомьтесь с книгой Дональда Кнута, "Искусство программирования".

Если Вас заинтересовала тема больших чисел, таких как гугол, ниже приводится перечень нескольких Web-страниц, которые настоятельно рекомендуется исследовать. Некоторые из приведенных там алгоритмов поддаются воплощению на языке Python.

1. Большие конечные и бесконечные числа: <http://www.sci.wsu.edu/math/faculty/hudelson/mose.r.html>
2. Познакомьтесь с числом газиллион (*gazillion*): <http://www.straightdope.com/mailbag/mgazilli.html>

3. Как получить число
гуголплекс: <http://WWW/~fp/Tools/GetAGoogol.html>
4. Названия чисел разного порядка (от маленьких до больших): <http://studwww.rug.ac.be/~hvernaev/FAQ/node26.html>
5. И снова число
гуголплекс: <http://www/~fp/Topls/Googool.html> (лучший узел, касающийся этого вопроса)
6. Число π и его
друзья: <http://www.go2net.com/internet/useless/useless/pi.html>
7. Эпоха расцвета индейцев Майя и эпоха
глифов: <http://www.pauahtun.org/calglypn.html>

Попробуйте выяснить, насколько большим должно быть число, чтобы вызвать сбой в работе Python или "подвесить" Ваш компьютер? (Только сначала сделайте резервные копии всех файлов.)

4-й час

Переменные и управление ими

В данной главе Вы познакомитесь с переменными и узнаете, каким образом их можно использовать. Мы научимся создавать такие умные программы на языке Python, что они сами будут анализировать переменные и принимать решения, что делать дальше. Например, мы научимся давать команду компьютеру выполнять одну и ту же операцию много раз до тех пор, пока значение определённой переменной не станет истинным или ложным. Пожалуй, это одна из тех задач, которую компьютеры выполняют чаще всего, и мы изучим, как это делается в Python. После овладения всеми этими средствами программирования мы применим наши знания для написания небольшой, но полезной программы.

Переменные

Возможно, Вы ещё помните, что такое переменные. Все эти иксы и игреки из курса алгебры средней школы, те самые, что изводили Вас, когда Вы пытались безуспешно заснуть на уроке в классе. Может быть, кто-то до сих пор нервно вздрагивает при мысли о необходимости решить уравнение. Переменные входят в состав большинства математических формул. Формула предоставляет Вам в общем виде схему решения задачи. Но чтобы найти ответ, необходимо вместо переменных подставить реальные значения и выполнить соответствующие математические действия.



Переменные в математических формулах и компьютерных программах служат одной и той же цели. Они олицетворяют собой всего лишь место, в которое можно поместить некоторое фактическое значение. Правда, в алгебре переменные являются вместилищем только числовых значений; тогда как в компьютерных программах переменные могут содержать информацию любого типа. Причем они могут хранить их все то время, пока работает ваша программа, или всего лишь несколько миллисекунд, если Вам этого достаточно. Переменные можно также использовать в качестве "бумаги для заметок", т.е. просто как место для записи промежуточного значения, чтобы оно не забылось, пока Вы будете работать над другой частью формулы или задачи. Если Вам легче представить переменные в виде шпаргалок, то так и поступайте.

Переменные в Python могут быть названы любым понравившимся Вам именем, лишь бы только оно не совпадало с зарезервированным ключевым словом Python. Имена в Python всегда должны начинаться обязательно с буквы или символа подчёркивания (`_`). Допустимыми именами переменных являются `i`, `z1` и `old`, а вот примеры недопустимых имён – `123abc` и `if`. В то же время следует учитывать, что некоторые слова лучше других подходят в качестве имён переменных, а в определённых контекстах некоторые имена значительно предпочтительнее других. Немного позже я подробнее разъясню смысл этих утверждений. Пока запомните такое общее положение, что лучше давать своим переменным такие названия, которые несут в себе определённое смысловое значение. Иногда лучше дать переменной длинное имя, если сложно подобрать осмысленную аббревиатуру.

Зарезервированные слова, называемые ключевыми, или служебными, – это слова, которые Python резервирует для собственных нужд, а посему никому непозволительно использовать их как-либо иначе, кроме способов, оговоренных в документации на Python. В перечень таких слов входят, например, имена инструкций `if`, `for` и многие другие. Полная таблица, в которой отображены все зарезервированные слова, приведена в конце этой главы.

А теперь запустите интерпретатор, и мы попробуем на практике применить некоторые переменные (рис. 4.1).

Обратите внимание, что когда Вы работаете с IDLE или Python в окне терминала, можно просто ввести имя переменной и нажать клавишу `<Enter>`, чтобы увидеть её значение. А если Вы используете Python в пакетном режиме для запуска и выполнения программ, у Вас такой фокус не пройдет. Для

вывода переменной необходимо будет воспользоваться инструкцией `print`. Инструкция `print` может использоваться несколькими способами. Все они будут описаны немного позже. А сейчас просто запомните, что в диалоговом режиме (IDLE) инструкции `print имя_переменной` и просто `имя переменной` ведут себя одинаково.

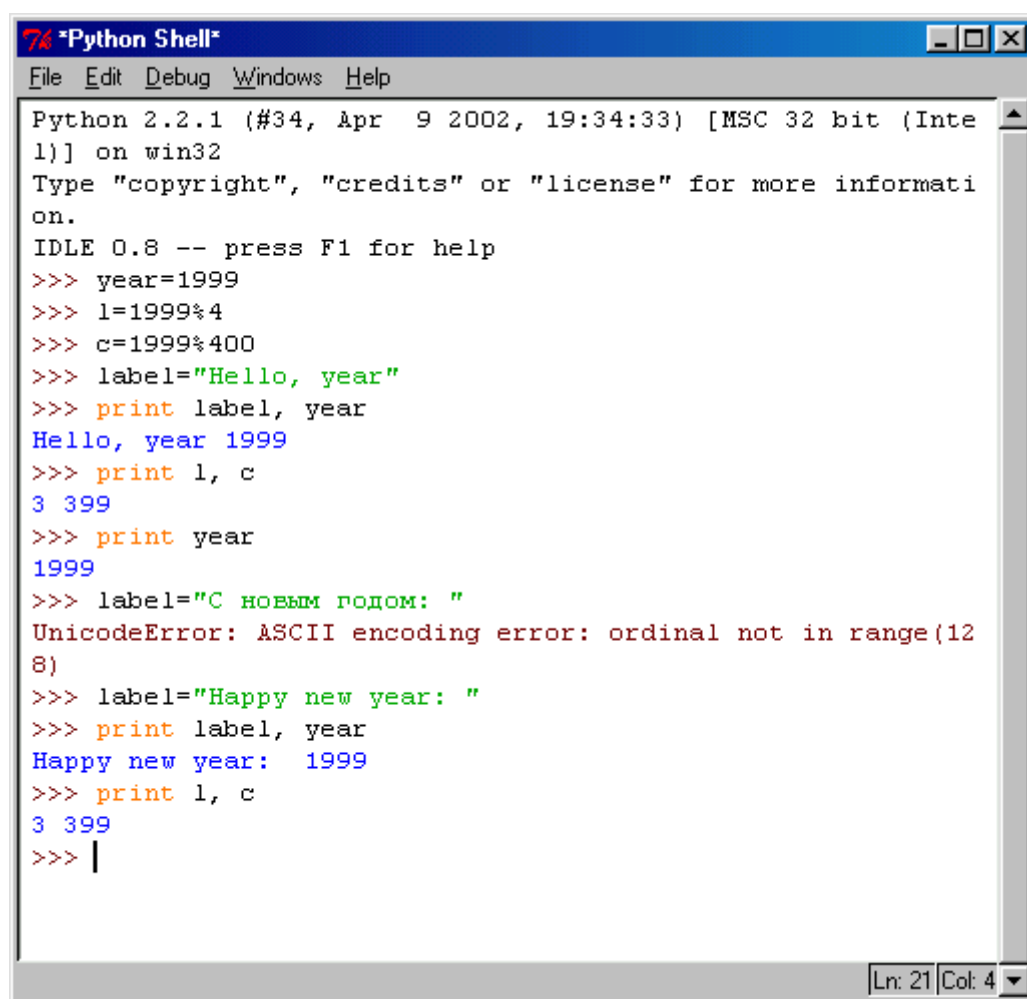
Рис. 4.1. Примеры переменных

Строка `googol=10**100L` показывает нам, что в качестве правого операнда может выступать целое выражение. Выражение — это часть кода, выполняющая некоторую работу. В данном случае выражение `10**100L` вычисляет значение 10^{100} , а символ

[illegible]

***Прим. В. Шипкова: такое вольное обращение с переменными допускает далеко не каждый язык программирования. Даже последние реализации Бейсика настоятельно рекомендуют делать предварительные объявления типов переменных. В этом есть как сильные, так и слабые стороны.**

переменных, перечислив их через запятые, непосредственно числовые значения, называемые константами, и даже строки текста (рис. 4.3).



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> year=1999
>>> l=1999%4
>>> c=1999%400
>>> label="Hello, year"
>>> print label, year
Hello, year 1999
>>> print l, c
3 399
>>> print year
1999
>>> label="С новым годом: "
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> label="Happy new year: "
>>> print label, year
Happy new year: 1999
>>> print l, c
3 399
>>> |
```

Рис. 4.3. Использование инструкции `print`

***Прим. В. Шипкова:** внимательный читатель уже обратил внимание на тот факт, что символы с русской кодировкой вызвали ошибку программы. Я сразу подскажу как решить эту проблему (далее по тексту её решение не приводится): на самом деле Python очень дружелюбен к национальным кодировкам.

Для использования в консоли, в начале файла указать:
-*- coding: cp1251 -*-

Для DOS-кодировки:
-*- coding: cp866 -*-

Для использования в Windows (и в IDLE в том числе):
-*- coding: utf8 -*-

IDLE позволяет настроить эти параметры. Кроме всего прочего у каждой переменной (строковой) есть метод для

перекодировки из одной кодовой таблицы в другую, пример:

```
a="Hello, World!"  
b=a.encoding("utf8")
```

В качестве кодировки может быть несколько десятков значений.

Безусловно, переменные являются чрезвычайно важным средством в наборе инструментария программирования, но чтобы выполнять действительно полезные вещи, Вы нуждаетесь кое в чём ещё. Прежде всего, Вам необходимо знать, как сообщить компьютеру, что в какой-то момент времени необходимо выполнить определённое действие и в зависимости от результата выбрать дальнейшую последовательность действий. Принятие программой правильных решений — это тема оставшейся части данной главы. Чтобы программа работала так, как Вам нужно, следует разработать логику выполнения программы. Для управления логикой выполнения программы в Python используется несколько инструкций, осуществляющих ветвление программы. Мы начнём с наиболее важной и часто используемой инструкции `if`.

Инструкции `if`, `elif` и `else`

Инструкция `if` является основным средством управления логикой выполнения программы. Синтаксис этой инструкции в Python очень прост:

```
if условие:  
    выражения
```

Компонент `условие` практически всегда содержит оператор сравнения. Таблица всех операторов сравнения Python приводится в конце этой главы в разделе "Практикум". Один из них — это оператор равенства (`==`). Обычно он используется для проверки того, что какая-то переменная равна некоторому значению, как показано в следующем примере:

```
if i == 1:  
    выражения
```

На русский язык приведенную выше проверку можно перевести примерно так: "Если переменная `i` равняется значению 1, то необходимо выполнить блок выражений". Блок выражений может содержать любое число строк со всеми возможными операторами

и инструкциями, в том числе и другие инструкции `if`. В других языках программирования допускается, чтобы компонент условие был представлен целым выражением, что позволяло бы в строке с инструкцией `if` одновременно выполнять вычисление, присвоение и проверку результатов. Но Guido посчитал предоставление таких возможностей нецелесообразным. Например, если позволить программистам использовать в одной строке с инструкцией `if` оператор присвоения, то это приведёт к тому, что станут допустимыми выражения следующего типа:

```
if i = 1:  
    выражения
```

В результате переменной `i` присваивается значение 1, а затем выполняется проверка истинности переменной. Понятно, что результат проверки всегда будет истинным, так как переменная `i` всегда будет равна значению 1. В этом примере программист скорее всего просто случайно вместо оператора равенства (`==`) ввёл оператор присвоения (`=`), что является довольно распространённой ошибкой. При программировании на языке C часто тратится много времени на поиск ошибок подобного типа. Guido решил, что время, потраченное на отладку программы, лучше бы было посвятить программированию, поэтому в Python в строке за инструкцией `if` переменные можно только проверять, но не изменять.

Блок выражений, следующих за инструкцией `if`, должен иметь отступ. В большинстве других языков программирования отступ является необязательным и служит лишь для улучшения читабельности кода. В Python отступ является составной частью языка и используется вместо специальных символов, которые в других языках устанавливают начало и конец блока. В языке Pascal, например, используется пара ключевых слов `begin` и `end`, поэтому для него конструкция с инструкцией `if` выглядит следующим образом:

```
if b<0 begin  
    result:=result+pi;  
end;
```

Символ `<` обозначает меньше чем, тогда как символ `>` обозначает больше чем. Тот же самый код в языке C выглядел бы примерно так:

```
if(b<0)  
{  
    result=result+pi;
```



```
}
```

или вот так:

```
if (b < 0) result = result + pi;
```

Обязательный отступ в Python — одна из наиболее спорных особенностей языка. По этому вопросу программисты разделились на два фронта. Одни горячо поддерживают это нововведение, другие ненавидят его. Поклонников языка Python, которых обязательное наличие отступа совершенно не трогает, меньше всего (я знал только одного — технического редактора американского издания этой книги). Но факт остаётся фактом, жаркие дискуссии о достоинствах и недостатках языка Python, которые иногда похожат на военные баталии, чаще всего ведутся вокруг использования отступов как составной части языка. Лично я нахожу это решение интересным и полезным, так как Python тем самым прививает программистам навыки составления удобочитаемых кодов. Кстати, также считает и Guido, поэтому он пообещал, что во всех последующих версиях Python обязательное использование отступов сохранится. Поэтому если Вы хотите овладеть программированием на Python, то с этой его особенностью Вам придётся смириться, нравится она Вам или нет.

Когда я ещё только изучал основы программирования, в комментариях к программам и в телеконференциях часто встречался термин iff. В течение длительного времени это сбивало меня с толку, потому что я не мог понять, частью какого языка программирования была эта инструкция. Но однажды я натолкнулся на комментарий в одной программе, который гласил: "iff. это if и ничего кроме if."

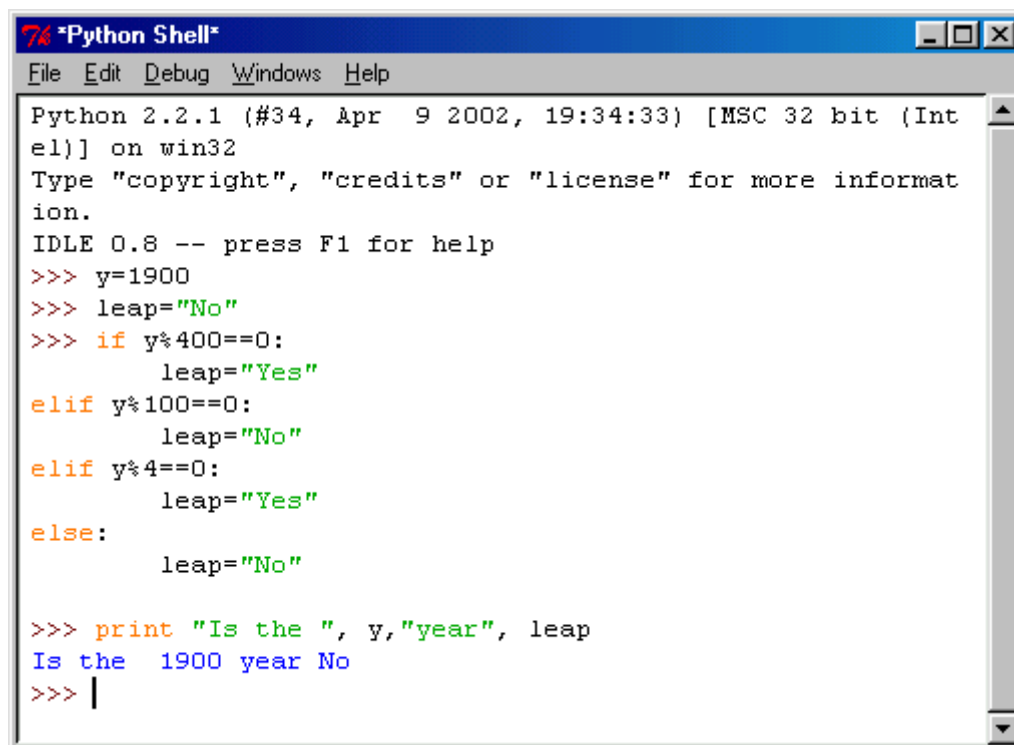
Полный синтаксис инструкции if предоставляет больше возможностей по управлению логикой выполнения программы:

```
if условие:
    выражения
elif условие:
    выражения
elif условие:
    выражения
else:
    выражения
```

Блоки выражений могут быть любой длины и включать любые допустимые инструкции Python. Сейчас мы посмотрим, как все это работает, поэкспериментировав с этими инструкциями в IDLE. Помните григорианское и юлианское правило определения



високосного года, о котором мы говорили в предыдущей главе? Запускайте IDLE. На рис. 4.4 показан полный вариант конструкции с инструкцией `if`. Данная программа используется для определения високосных годов в григорианском календаре.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> y=1900
>>> leap="No"
>>> if y%400==0:
    leap="Yes"
elif y%100==0:
    leap="No"
elif y%4==0:
    leap="Yes"
else:
    leap="No"

>>> print "Is the ", y, "year", leap
Is the 1900 year No
>>> |
```

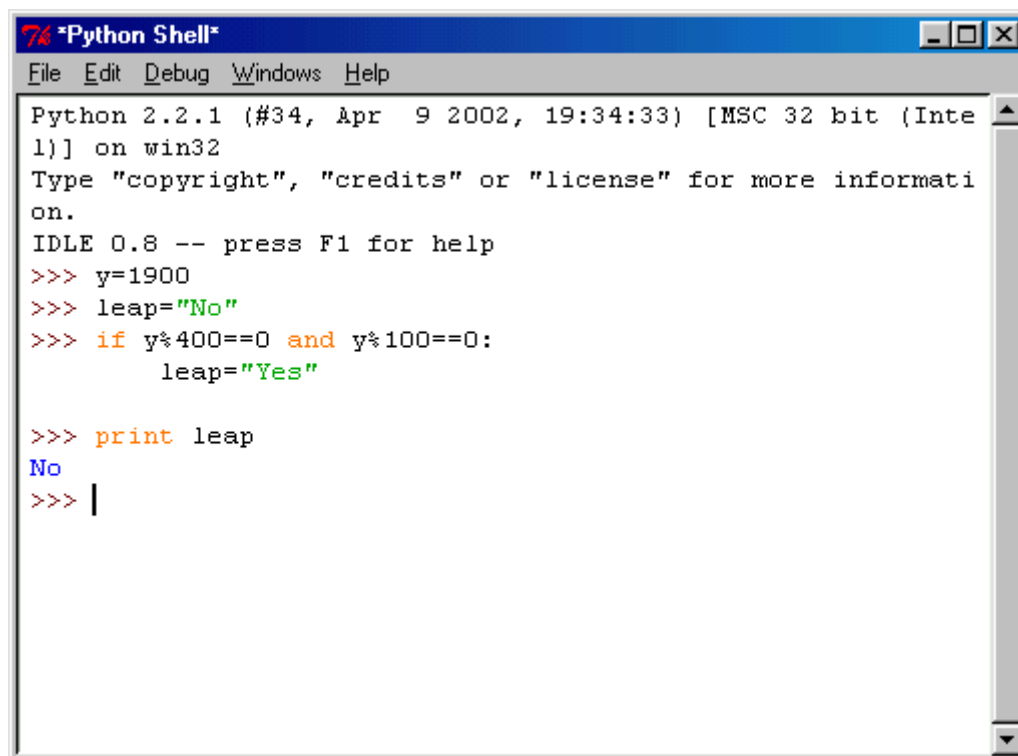
Рис. 4.4. Инструкция `if` и григорианский календарь

Обратите внимание, как редактор IDLE указывает завершение работы из кода блока программы. Для блока автоматически задается отступ, который хорошо виден на рис. 4.4. Нажатие клавиши `<Enter>` после пустой строки блока означает, что работа над блоком закончена. Редактор IDLE автоматически отменит отступ текущего блока и вернет курсор к началу строки (как на рис. 4.4) или к отступу предыдущего блока. Интерпретатор будет вести себя аналогично при работе в режиме командной строки, хотя отступы будут выглядеть несколько иначе.

Для юлианского календаря Вам нужно будет проверять только условие делимости на 4. В юлианском календаре 1900 год был бы високосным.

В условии инструкции `if` допускается выполнять несколько проверок. Для этого используются специальные логические операторы `and`, `or` и `not`. Если в одну инструкцию `if` поместить две проверки условий, объединённые оператором `and`, то `if` возвратит `true` (истинно) только в том случае, если будут истинными оба условия. Если первая проверка возвратит значение `false` (ложно), то вторая проверка выполняться уже не будет. При использовании оператора `or`

истинной может быть любая из проверок. Если первая проверка возвращает `true`, то вторая проверка проводиться не будет, поскольку в ней уже нет смысла. На рис. 4.5 показано объединение двух условий в инструкции `if`.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
IDLE 0.8 -- press F1 for help
>>> y=1900
>>> leap="No"
>>> if y%400==0 and y%100==0:
        leap="Yes"

>>> print leap
No
>>> |
```

Рис. 4.5. Объединение двух условий с помощью оператора `and`

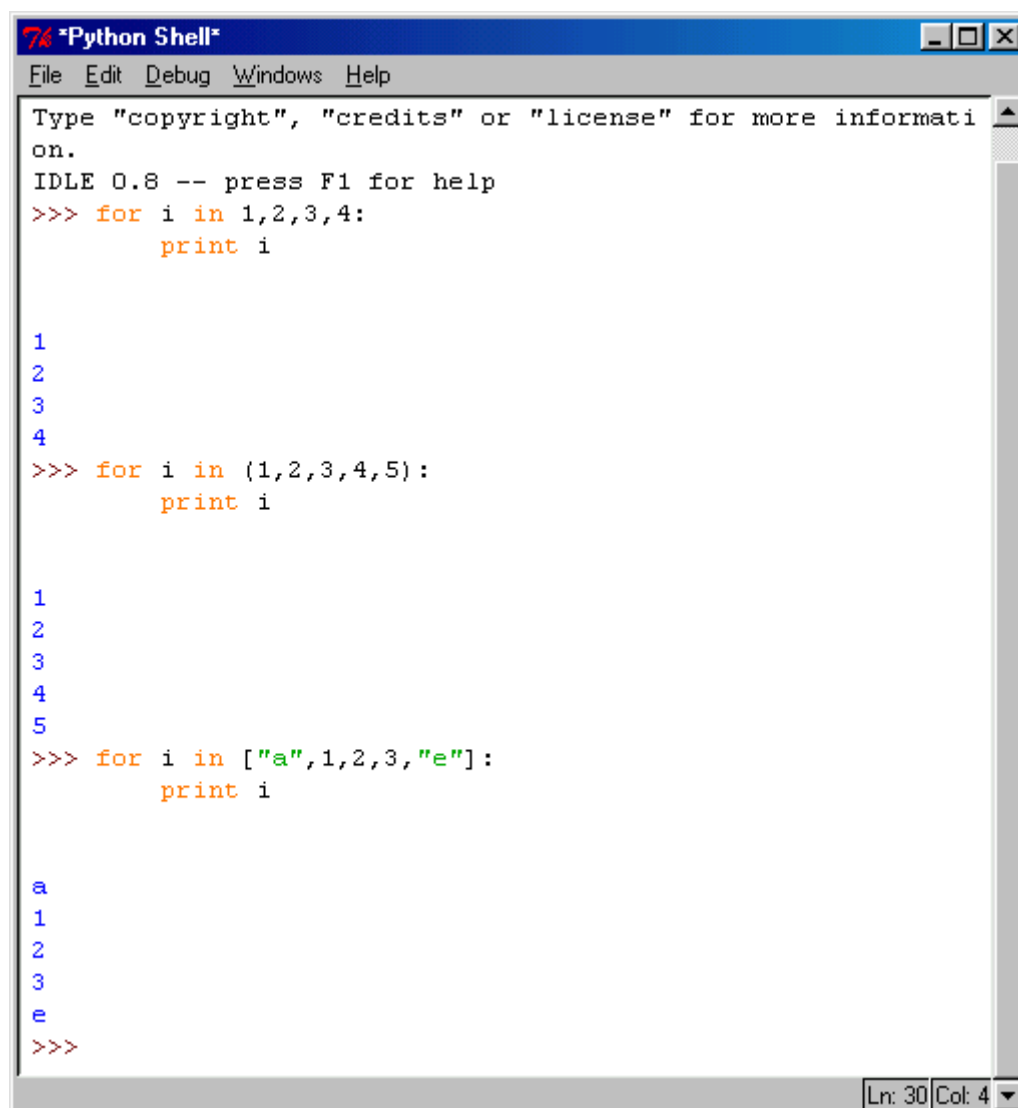
В приведенном примере сначала проверяется условие делимости на 400, и эта проверка возвращает значение `false`. Поэтому проверка делимости на 100 уже не выполняется.

Инструкция `for` и функция `range()`

Хотя `if` — чрезвычайно полезная инструкция, с её помощью можно решить далеко не все проблемы с управлением логикой выполнения программы. Например, она бесполезна в тех случаях, когда нужно выполнить какой-нибудь программный блок много раз. В Python имеется инструкция `for`, которая идеально подходит для решения этой задачи. С её помощью можно повторить выполнение блока требуемое число раз, хотя отсчёт циклов происходит несколько иначе, чем в других языках программирования. В других языках для программирования числа циклов используются специальные переменные, определяющие условия начала и завершения цикла. Определение этих условий может быть связано со сложными вычислениями. Инструкция `for` языка Python работает проще. Ей предоставляется список значений, которые поочередно присваиваются целевой переменной, используемой в блоке выражений, следующем за инструкцией `for`. Вот синтаксис применения этой инструкции:

```
for целевая_переменная in список:  
    выражения  
else:  
    выражения "выполняются только в том случае,  
    если цикл не был прерван инструкцией break"
```

Блок выражений, следующий за инструкцией `for`, называется *телом цикла*. В теле инструкции `for` можно использовать инструкцию прерывания `break`, которая делает именно то, что следует из её названия, – прерывает выполнение цикла. Выражения, следующие за `else`, выполняются только в том случае, если цикл не был прерван инструкцией `break`. Как правило, выполнение этой части кода случается крайне редко. Как знамение Божье, выполнение данных выражений должно сигнализировать о совпадении неких уникальных событий. Более подробно мы изучим инструкцию `break` в следующем разделе, когда перейдем к рассмотрению конструкций с инструкцией `while`. А пока взгляните на рис. 4.6, где показан пример применения инструкции `for`.



```
Python Shell
File Edit Debug Windows Help
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> for i in 1,2,3,4:
    print i

1
2
3
4
>>> for i in (1,2,3,4,5):
    print i

1
2
3
4
5
>>> for i in ["a",1,2,3,"e"]:
    print i

a
1
2
3
e
>>>
```

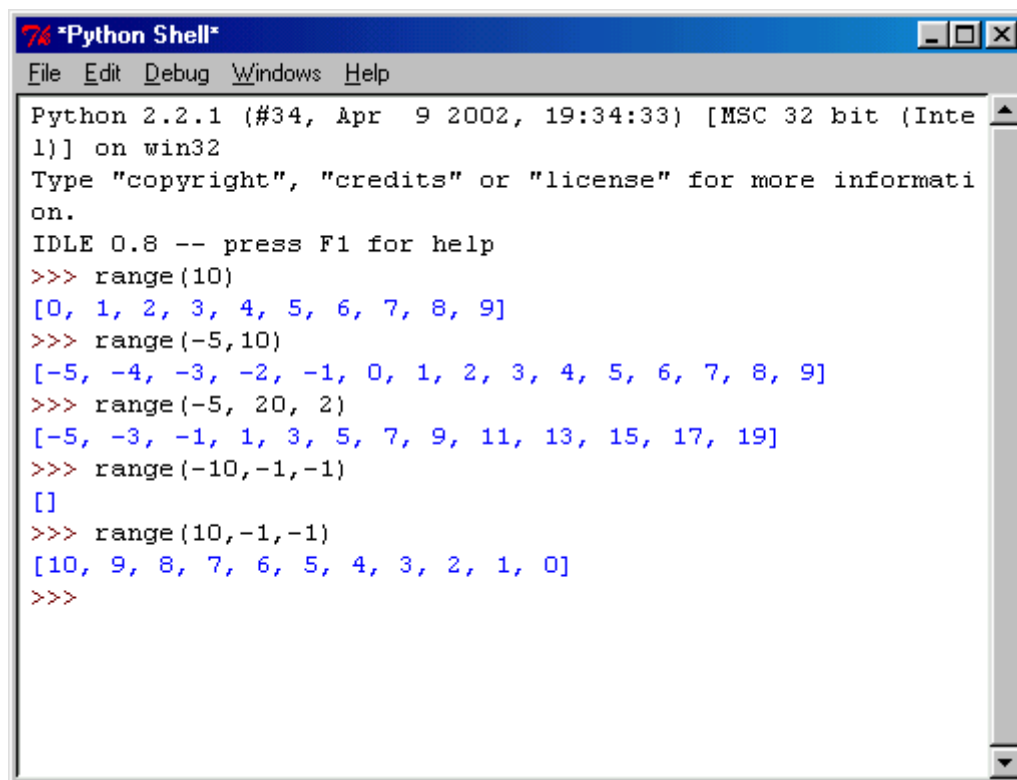
Рис. 4.6. Инструкция `for`



Как показано в примерах на рис. 4.6, компонент `список` действительно является списком некоторых значений, представленных с использованием следующих трёх допустимых синтаксисов: `1,2,3,4`, `(1,2,3,4,5)` и `["a",2,3,4,"e"]`. Инструкция `for` поочерёдно на каждой итерации присваивает переменной `i` значения из списков и выполняет выражения тела цикла. Вы можете убедиться, что всё именно так и происходит, поскольку в нашем примере тело цикла содержит единственную инструкцию `print i`, которая выводит на экран текущее значение переменной `i`. Обратите внимание, что совершенно необязательно, чтобы в списке были только числа. Инструкция `for` одинаково хорошо справляется также с символами и целыми строками, выступающими элементами списка.

Термин итерация означает повторное выполнение чего-либо много раз. Таким образом, этот термин подходит для описания выполнения одного цикла программы, хотя в этом случае ещё иногда говорят о шаге цикла. Забивание молотком гвоздя в стену также можно назвать повторяющимся циклическим процессом. При этом каждый удар молотка будет итерацией. При успешном завершении всех циклов удара молотка Вы получите полезный результат – забитый в стену гвоздь. При неуспешном выполнении – побитые пальцы.

Очевидный недостаток показанного выше синтаксиса использования инструкции `for` – это необходимость вводить в программу список всех возможных значений целевой переменной, который может оказаться слишком длинным. Эту проблему можно решить с помощью функции `range()`. Функция `range()` – это ещё одна встроенная функция языка Python. (Термин *встроенная* означает, что Python всегда знает о ней, поэтому нет необходимости как-либо определять её в программе перед использованием.) Данная функция генерирует упорядоченные списки чисел. Можно использовать её четырьмя способами. Первый, и самый простой, показан в первой строке на рис. 4.7.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following text:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
IDLE 0.8 -- press F1 for help
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(-5,10)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(-5, 20, 2)
[-5, -3, -1, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> range(-10,-1,-1)
[]
>>> range(10,-1,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>>
```

Рис. 4.7. Функция `range()`

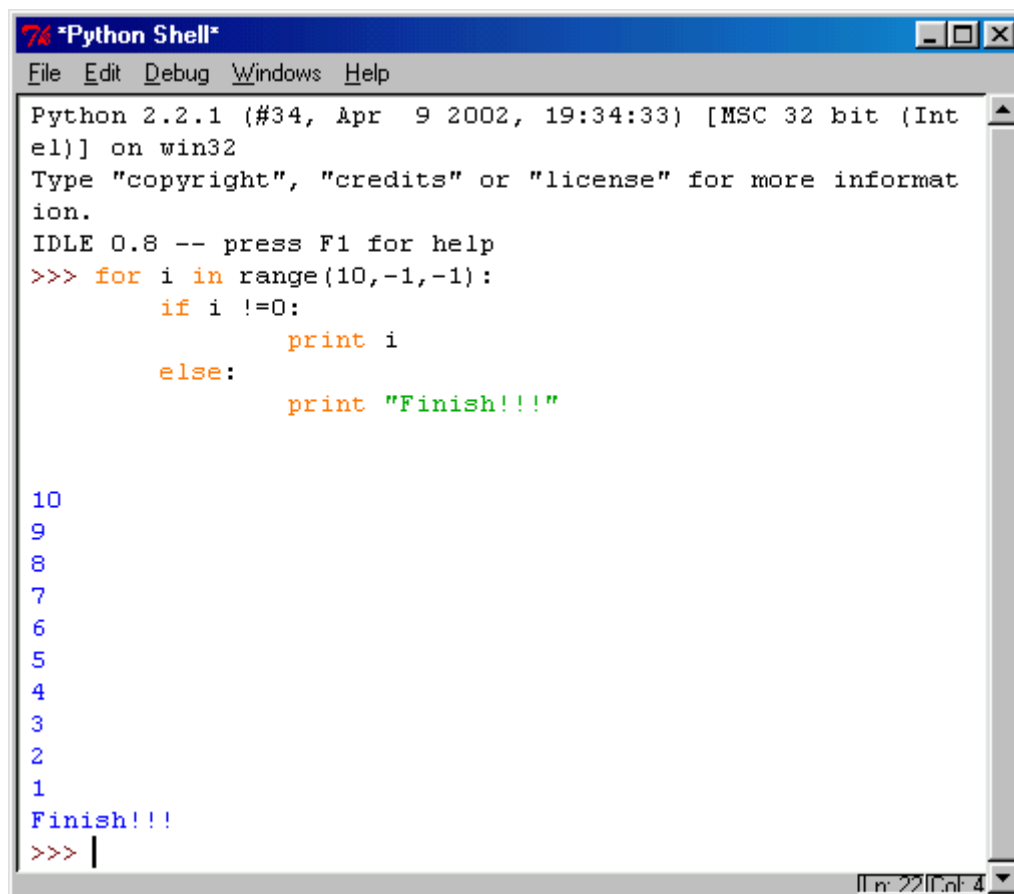
Первый вариант — `range(10)` — всего лишь указывает функции составить список, содержащий цифры от 0 до 9. Её единственный аргумент — это не размер списка, а последнее число списка, тогда как начальное значение всегда равно нулю. Как и в языке C, числовые последовательности начинаются с нуля, а заканчиваются (при положительном приращении) числом, которое на единицу меньше граничного значения. Например, если граничное значение равно 10, выход из цикла произойдет после 9. Интересно, что произойдет, если подставить отрицательное число? А что случится, если попытаться выполнить `range(0)`? Попробуйте сделать это сами.

Вторая форма, также показанная на рис. 4.7, — `range(-5,10)`. Функция должна составить список, который, как и в прежнем случае, заканчивается значением, на единицу меньше граничного (т.е. 9), но начинающийся с -5. Вот так можно получить список, который будет начинаться с отрицательного числа.

Третий вариант — `range(-5,20,2)` — показывает, что в общем случае существуют три допустимых аргумента этой функции: необязательный аргумент начала, обязательный аргумент окончания и вспомогательный параметр — шаг. Как видите, при использовании этой формы записи список начинается с -5, приращение для каждого следующего элемента составляет 2, а заканчивается список значением 19.

Наконец, четвёртый вариант демонстрирует, что `range()` не всегда должна следовать в положительном направлении. Если начальное значение больше конечного, а шаг отрицательный, то функция `range()` создаёт последовательность уменьшающихся значений.

В программах на языке Python часто используется комбинация функций и инструкций `for`, `range()` и `if`– На рис. 4.8 показан пример такой конструкции.



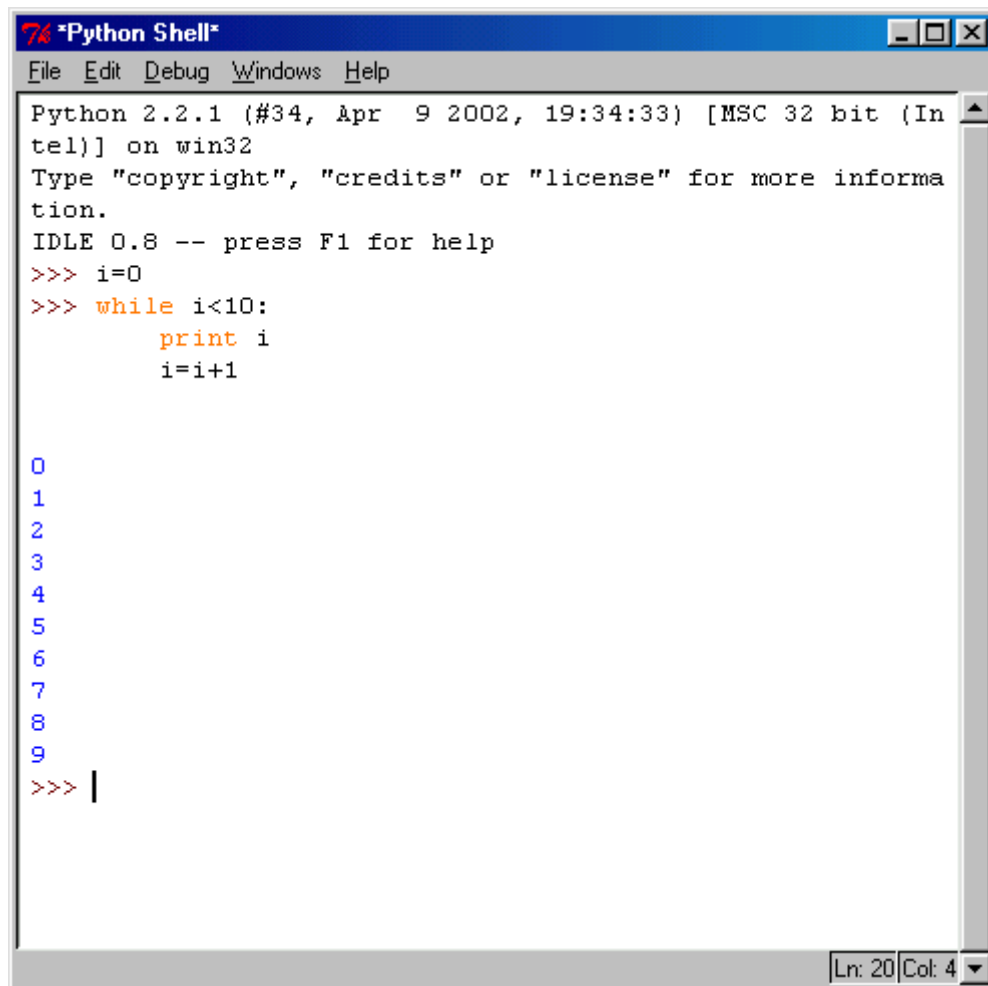
```
Python 2.2.1 (#34, Apr  9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> for i in range(10,-1,-1):
        if i !=0:
            print i
        else:
            print "Finish!!!"

10
9
8
7
6
5
4
3
2
1
Finish!!!
>>> |
```

Рис. 4.8. Конструкция с инструкциями `for`, `if` и функцией `range()`

Инструкция `while`

Инструкция `while`, также относится к числу тех, которые сообщают Python, что определённое действие необходимо выполнить несколько раз. Но в отличие от инструкции `for` она имеет другую синтаксическую конструкцию. Ее синтаксис показан на рис. 4.9.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following code:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> i=0
>>> while i<10:
>>>     print i
>>>     i=i+1
```

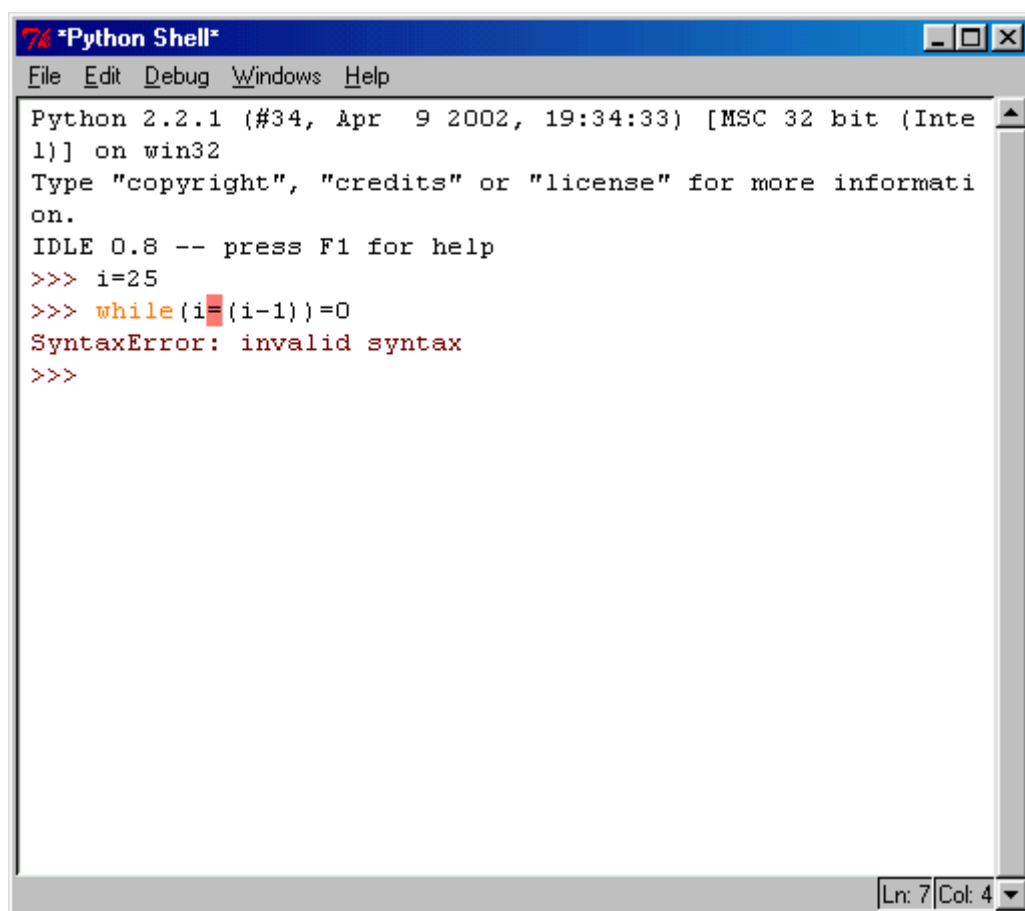
The output of the program is displayed on the left side of the window, showing the numbers 0 through 9, each on a new line. The status bar at the bottom right indicates "Ln: 20 Col: 4".

Рис. 4.9. Инструкция *while* с проверкой условия

Конечно, посчитать от 0 до 9 можно было и с помощью инструкции *for*. Но инструкцию *while* можно использовать не только для счёта. Ее назначение — отслеживать присутствие в списке особого элемента или выполнение какого-либо события. На рис. 4.10 показана первая из этих двух ситуаций.

Обратите внимание, что в этом примере показана ложная проверка условия *while* 1, всегда возвращающего *true*. Это означает, что до тех пор, пока Вы как программист не вмешаетесь в работу программы, инструкция *while* 1 будет выполняться вечно. Этот случай называется **бесконечным циклом**. Бесконечный цикл, который действительно выполняется вечно, — это одна из ошибок программирования. Но тот же бесконечный цикл может оказаться весьма полезным и эффективным решением в том случае, если Вы предусмотрите условие его прерывания. В условии инструкции *while*, как и в случае с инструкцией *if*, мы не можем присваивать значения переменным. На рис. 4.11 показана попытка присвоить значение переменной *i* в условии инструкции *while*, что завершилось показом сообщения об ошибке. Следовательно, заботу об изменении значения переменной *i* мы должны взять на себя. Именно поэтому в коде появилась строка *i = i - 1*.

И в заключение рассмотрим способ выхода из бесконечного цикла. Как правило, для этого используется комбинация инструкций `if` и `break`, в которой проверяется соответствие целевой переменной определённому значению и в случае обнаружения тождества прерывается цикл `while`.



The screenshot shows a window titled "Python Shell" with a menu bar (File, Edit, Debug, Windows, Help). The text area contains the following text:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> i=25
>>> while(i=(i-1))=0
SyntaxError: invalid syntax
>>>
```

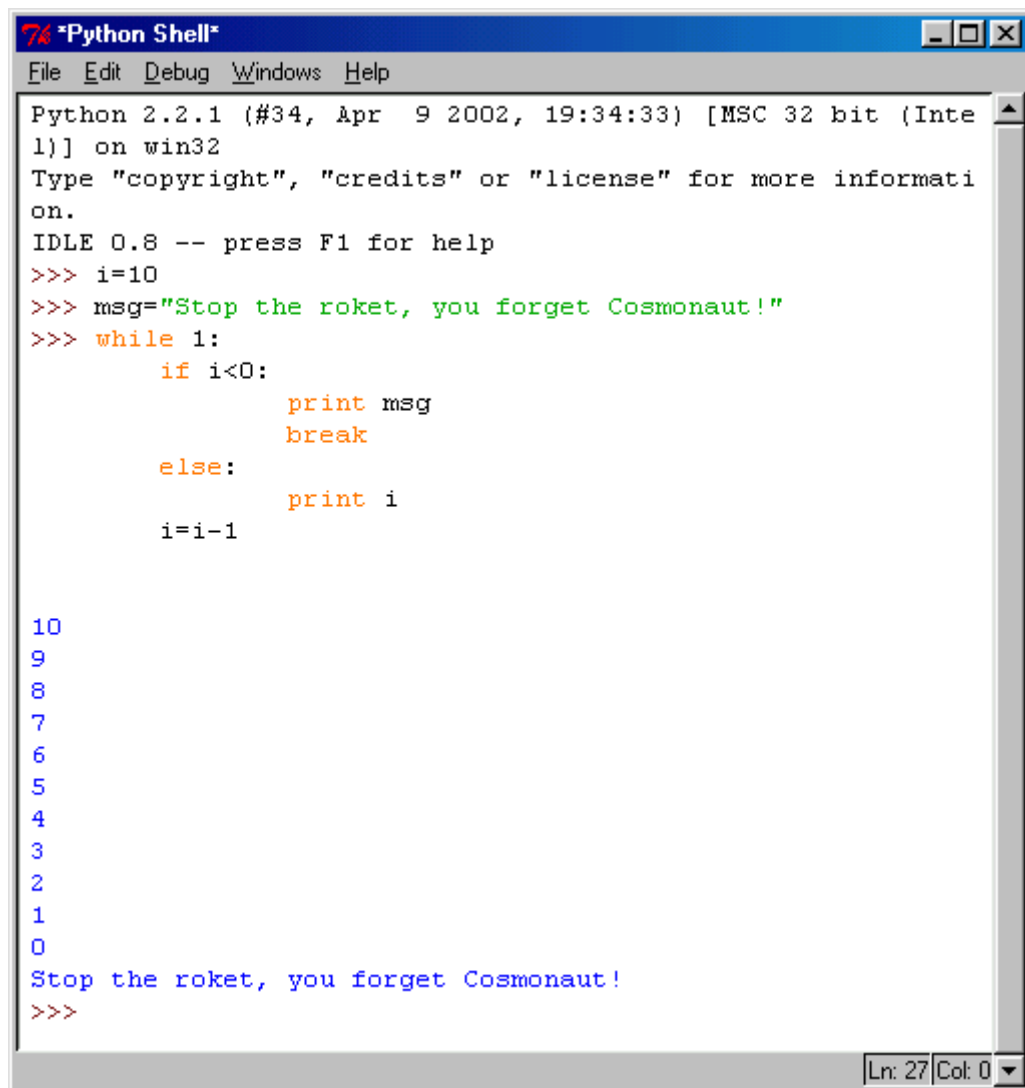
The status bar at the bottom right shows "Ln: 7 Col: 4".

Рис. 4.10. Бесконечный цикл с инструкцией `while`

***Прим. В. Шипкова:** на самом деле на рисунке сверху не изображён бесконечный цикл – на нём изображён НЕПРАВИЛЬНЫЙ цикл. Бесконечный цикл будет выглядеть так:

```
a=20
while a>=15:
    print "Das ist kaput!"
a=a-10
```

В указанном примере цикл `while` будет выполняться до тех пор, пока `a` больше или равно 15. Но в связи с тем, что значение `a` меняется только за пределами цикла – такой цикл будет действительно "вечным". Учитывая тот факт, что нет условия выхода из этого цикла – этот цикл является ошибкой программиста.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> i=10
>>> msg="Stop the roket, you forget Cosmonaut!"
>>> while 1:
    if i<0:
        print msg
        break
    else:
        print i
    i=i-1

10
9
8
7
6
5
4
3
2
1
0
Stop the roket, you forget Cosmonaut!
>>>
```

Рис. 4.11. Ошибка, связанная с попыткой присвоить значение в условии инструкции `while`

***Прим. В. Шипкова: а вот на этом рисунке как раз должен быть рисунок 4.10. %-)**

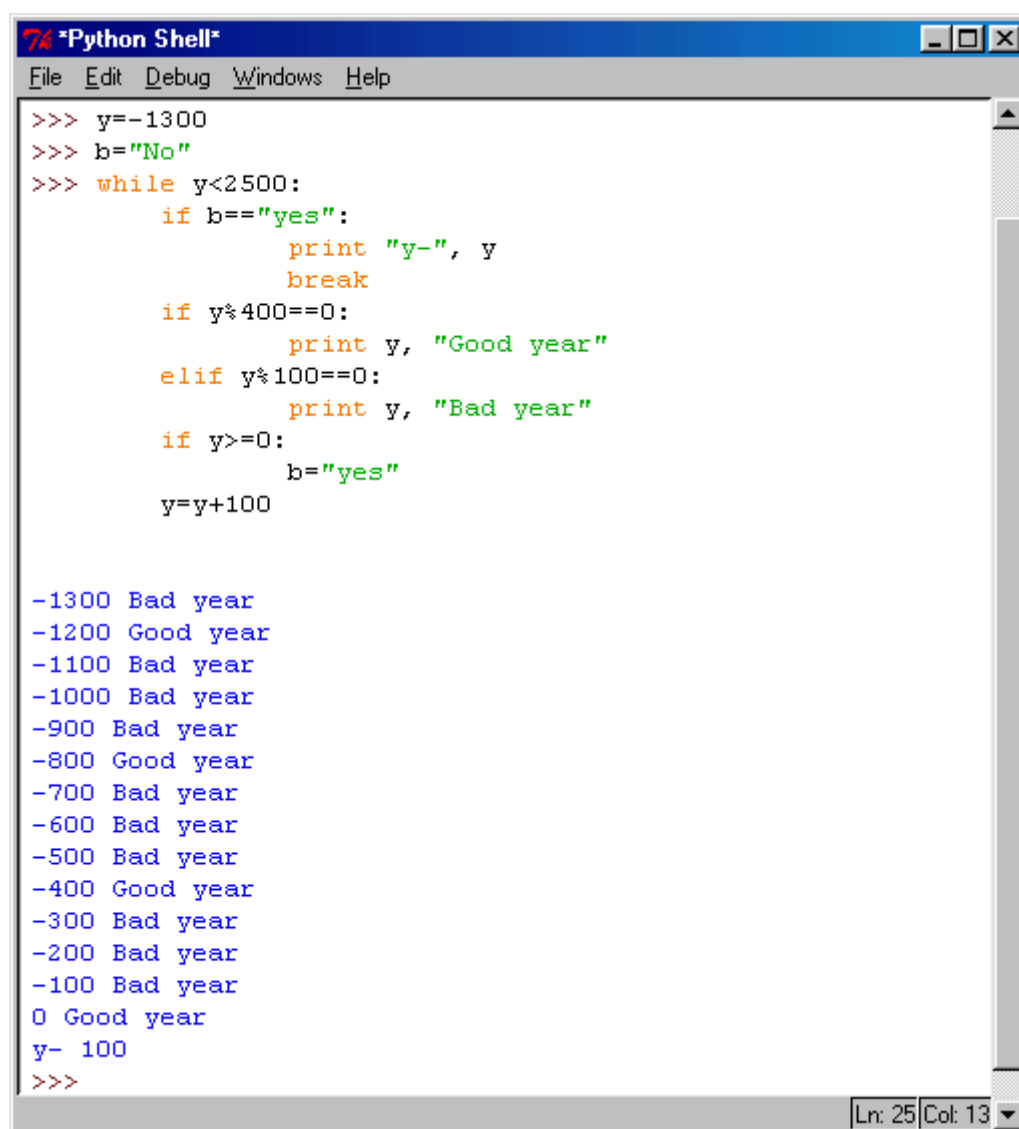
Полный синтаксис инструкции `while`, как и в случае с инструкцией `for`, содержит ещё блок `else`:

```
while условие:
    выражения
else:
    выражения "выполняются только в том случае,
    если цикл не бал прерван инструкцией break"
```

Опять-таки, блок инструкций за инструкцией `else` выполняется только при "естественном" завершении цикла `while`.

Другой вариант бесконечного цикла с инструкцией `while`, который будет выполняться до тех пор, пока не произойдет некоторое событие, представлен на рис. 4.12.

Действительно, данный пример выглядит несколько надуманно, но в следующем разделе мы найдем способ сделать данный код более естественным.



```
Python Shell
File Edit Debug Windows Help

>>> y=-1300
>>> b="No"
>>> while y<2500:
>>>     if b=="yes":
>>>         print "y-", y
>>>         break
>>>     if y%400==0:
>>>         print y, "Good year"
>>>     elif y%100==0:
>>>         print y, "Bad year"
>>>     if y>=0:
>>>         b="yes"
>>>     y=y+100

-1300 Bad year
-1200 Good year
-1100 Bad year
-1000 Bad year
-900 Bad year
-800 Good year
-700 Bad year
-600 Bad year
-500 Bad year
-400 Good year
-300 Bad year
-200 Bad year
-100 Bad year
0 Good year
y- 100
>>>
```

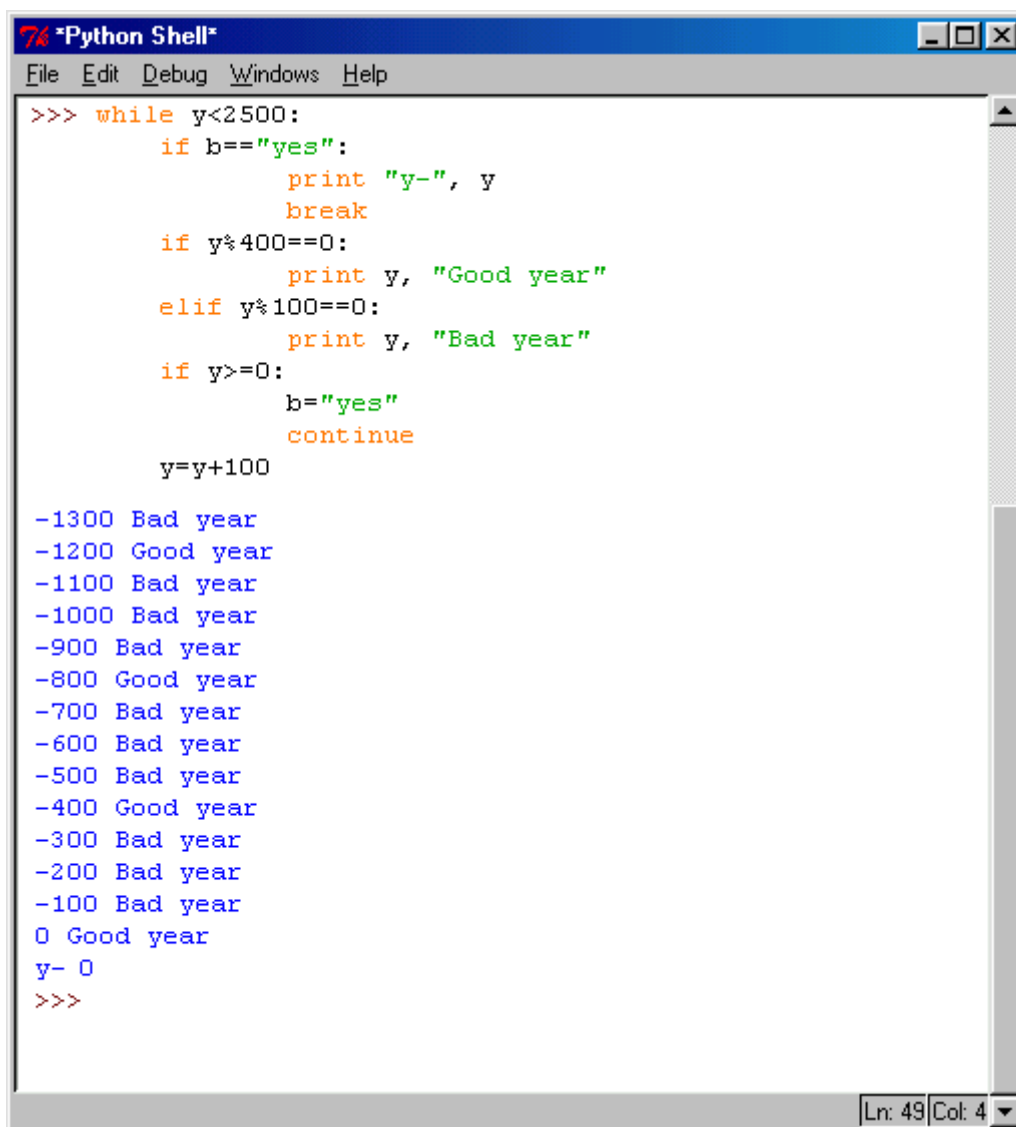
Рис. 4.12. ещё один пример бесконечного цикла с инструкциями `while` и `if`

***Прим. В. Шипкова: как видно, всё-таки этот цикл закончился без экстренных мер – этот цикл не является бесконечным.**

Инструкции `break`, `continue` и `pass`

Как Вы уже видели раньше, инструкция `break` используется для выхода из цикла, заданного инструкцией `for` или `while`. В других ситуациях Вы можете посчитать более целесообразным

не выходить из цикла полностью, а продолжить его выполнение начиная с первой строки тела цикла. Именно это выполняет инструкция `continue`. Когда в программе в теле какого-либо цикла встречается данная инструкция, выполнение программы немедленно переходит к самому первому выражению в теле цикла `for` или `while`, игнорируя все оставшиеся выражения в блоке. Таким образом, с помощью комбинации инструкций `continue` и `if` можно добиться того, чтобы перед завершением цикла программа выполняла специальный набор выражений, как показано на рис. 4.13. (Этот пример является модификацией предыдущего кода, выполнение которого показано на рис. 4.12. В данном случае результат выглядит более естественным.)



```
>>> while y<2500:
    if b=="yes":
        print "y-", y
        break
    if y%400==0:
        print y, "Good year"
    elif y%100==0:
        print y, "Bad year"
    if y>=0:
        b="yes"
        continue
    y=y+100

-1300 Bad year
-1200 Good year
-1100 Bad year
-1000 Bad year
-900 Bad year
-800 Good year
-700 Bad year
-600 Bad year
-500 Bad year
-400 Good year
-300 Bad year
-200 Bad year
-100 Bad year
0 Good year
y- 0
>>>
```

Рис. 4.13. Конструкция с инструкциями `while`, `break` и `continue`

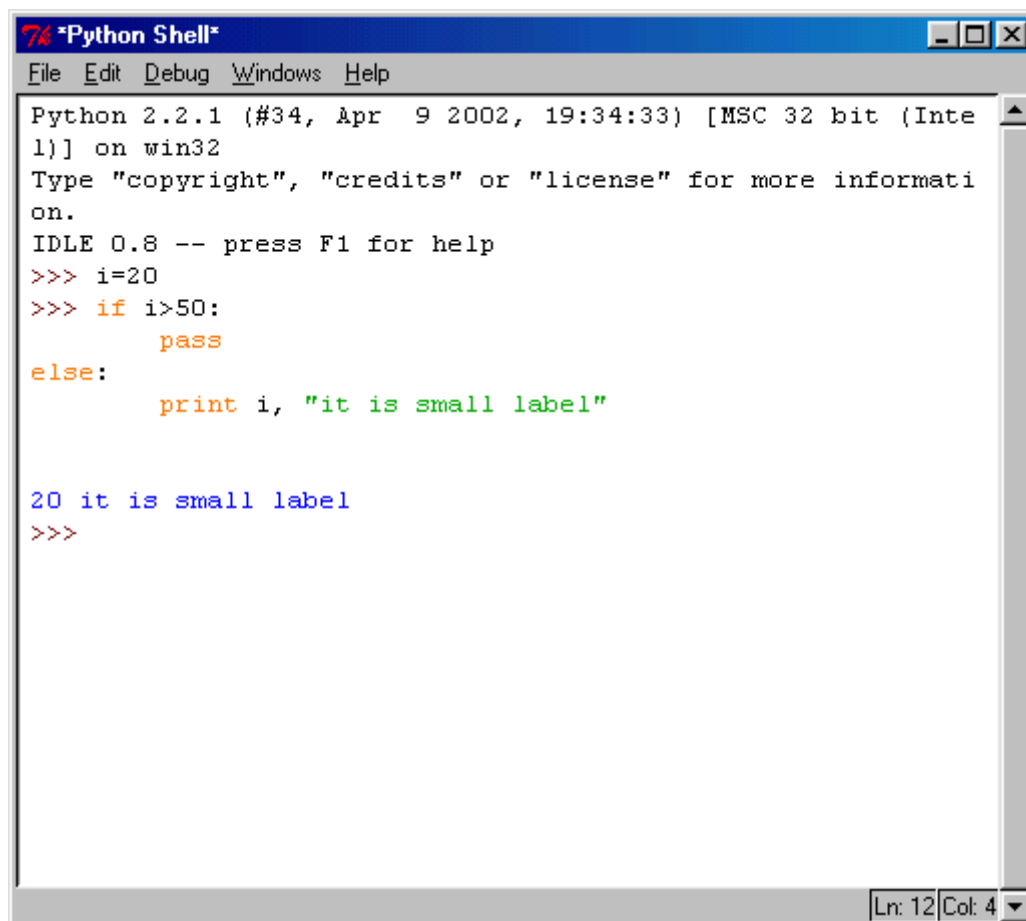
Единственное различие между этим примером и предыдущим состоит в том, что благодаря использованию инструкции `continue` к последнему значению переменной `y` не прибавляется значение 100. Окончательный результат в этом случае будет

равен нулю, тогда как в предыдущем примере цикл завершался со значением переменной *y*, равным 100.

В Python есть ещё одна важная инструкция: *pass*, которая, правда, не очень часто применяется в циклах *for* и *while*. В основном она используется в определениях классов, которые мы рассмотрим в части II этой книги. Инструкция *pass* применяется для указания Python, что в этом месте он ничего не должен делать. Такие инструкции в литературе называют *no-op* (от *no operation* – бездействие.) Может показаться весьма странным, что компьютеры нуждаются в подобной специальной инструкции, поскольку они только тем и занимаются, что выполняют то, что им явно укажут сделать (по крайней мере, так считают большинство людей). Но на заре создания компьютерных игр инструкции *no-op* часто использовались для создания игрового таймера. Принцип их использования заключался в том, что хотя по данной команде не выполняется никакая работа, но для её обработки компьютер все равно затрачивает определенное время. Поэтому, чтобы некоторое событие произошло вовремя, программисты обычно использовали циклы *for*, которые ничего не делали в течение заданного количества циклов. Таким образом, просто обеспечивалась требуемая задержка в выполнении программы, что позволяло управлять последовательностью событий в реальном времени. Инструкции бездействия типа *pass* уже не играют столь важной роли, как это было раньше, но иногда они все же смогут оказаться полезными. На рис. 4.14 показан пример использования инструкции *pass* совместно с инструкцией *if*. Если удалить строку с инструкцией *pass*, то код также будет выглядеть логичным:

```
if test:
else:
    do something
```

Но в этом случае Python покажет сообщение об ошибке (если- хотите, проверьте). Представленный выше синтаксис является недопустимым в Python. Правильным будет написать так, как показано на рис. 4.14.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
IDLE 0.8 -- press F1 for help
>>> i=20
>>> if i>50:
>>>     pass
>>> else:
>>>     print i, "it is small label"

20 it is small label
>>>
```

Рис. 4.14. Конструкция с инструкциями *if* и *pass*

Завершённая программа

Как и было обещано, ниже приведён листинг завершённой программы, в которой используются средства программирования, рассмотренные в данной главе.

Листинг 4.1. Завершённая программа с использованием средств управления логикой выполнения

```
#!/c:/python/python.exe
# верхнюю строку совсем не обязательно указывать
import sys
import string

if len(sys.argv)<2:
    print "Usage: leap.py year, year, year."
    sys.exit(0)

for i in sys.argv[1:]:
    try:
        y=string.atoi(i)
    except:
        print i,"is not a year."
```

```

continue
    leap="no"
    if y%400==0:
        leap="yes"
    elif y%100==0:
        leap="no"
    elif y%4==0:
        leap="yes"
    else:
        leap="no"

    print y, "leap:", leap, "in the Gregorian calendar"

if y%4==0:
    leap="yes"
else:
    leap="no"

print y, "leap:", leap, "in the Julian calendar"

print "Calculated leapness for", len(sys.argv) - 1, "years"

```

Вы можете набрать эту программу в текстовом редакторе и сохранить в файле под названием leap.py. Запустите её на выполнение, введя в командной строке DOS или оболочки UNIX `python leap.py 1900 1904 2000`. Затем попытайтесь ввести данные, не являющиеся годом, например `abcde`. Посмотрите, как отреагирует программа на попытку обмануть её. Немного позже Вы познакомитесь с этой странной записью `[1:]`, а также с функцией `len()` и другими операторами и инструкциями, которых прежде ещё не встречали.

Резюме

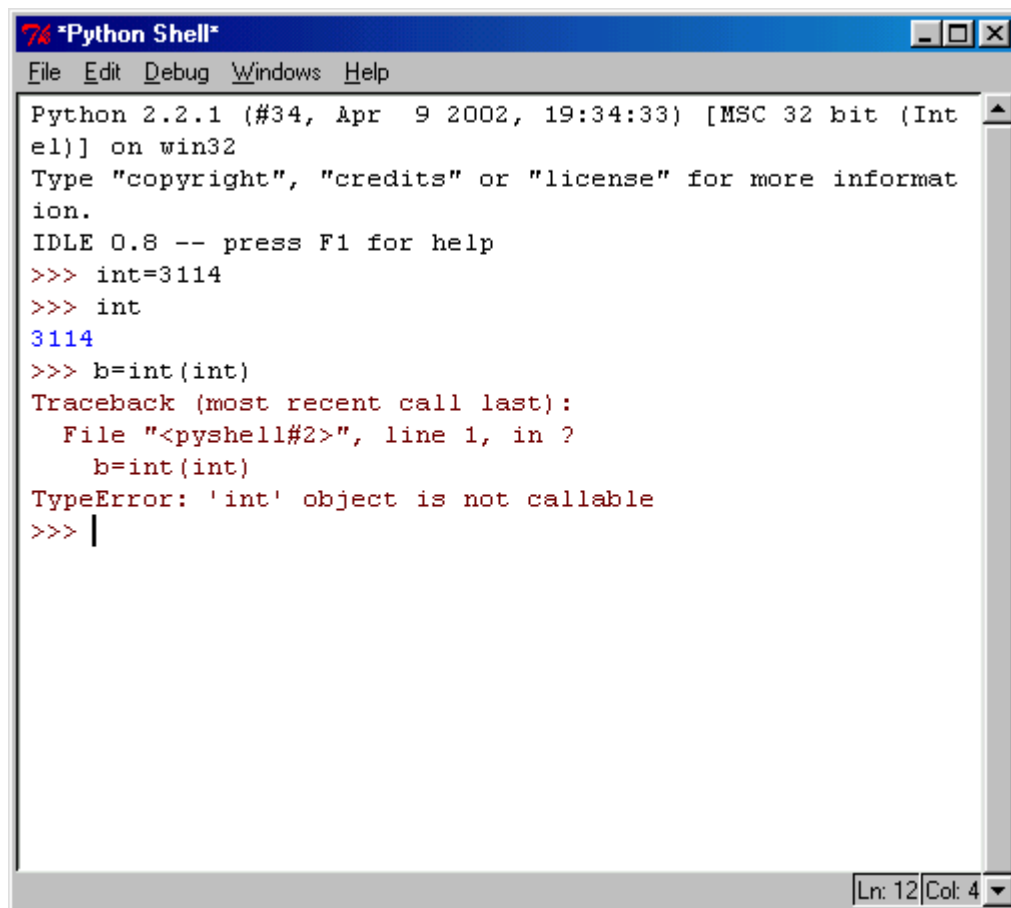
В ходе рассмотрения переменных в Python Вы узнали, что некоторые слова в этом языке зарезервированы для специальных целей. В табл. 4.1 приводится список всех ключевых слов языка Python.

Таблица 4.1. Ключевые слова в Python

and	elif	global	or
assert	else	if	pass
break	except	import	print
class	exec	in	raise
continue	finally	is	return
def	for	lambda	try

del**from****not****while**

Примите к сведению, что в Python допускается использовать названия функций в качестве имён переменных. Другими словами, вполне возможно использовать `int` в качестве имени переменной, хотя это слово является названием встроенной функции `int()`. Но так поступать не стоит. На рис. 4.15 показано, что может произойти, если непродуманно поступать подобным образом.



The screenshot shows a 'Python Shell' window with the following text:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> int=3114
>>> int
3114
>>> b=int(int)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in ?
    b=int(int)
TypeError: 'int' object is not callable
>>> |
```

At the bottom right of the window, it says 'Ln: 12 Col: 4'.

Рис. 4.15. В Python не всё разрешено, что не запрещено

Имена функций в Python являются всего лишь переменными особого типа и не защищены от неправильного употребления.

В этой главе Вы также научились создавать конструкции с инструкцией `if`. В условии инструкции `if` используются операторы сравнения, полный список которых приведён в табл 4.2.

Таблица 4.2. Операторы сравнения в Python

<code>a < b</code>	Истинно, если <code>a</code> меньше <code>b</code>	<code>a <= b</code>	Истинно, если <code>a</code> меньше или равно <code>b</code>
------------------------------	---	-------------------------------	---

<code>a > b</code>	Истинно, если <code>a</code> больше <code>b</code>	<code>a >= b</code>	Истинно, если <code>a</code> больше или равно <code>b</code>
<code>a == b</code>	Истинно, если <code>a</code> равно <code>b</code>	<code>a != b</code>	Истинно, если <code>a</code> не равно <code>b</code>
<code>a is b</code>	Истинно, если <code>a</code> тождественна <code>b</code>	<code>a is not b</code>	Истинно, если <code>a</code> не тождественна <code>b</code>
<code>a < b < c</code>	Истинно, если <code>a</code> меньше <code>b</code> и <code>b</code> меньше <code>c</code> . (Возможна аналогичная комбинация с оператором >)	<code>a and b</code>	Если <code>a</code> равно 0 или None, использовать <code>a</code> , иначе использовать <code>b</code>
<code>not a</code>	Истинно, если <code>a</code> равно 0 или None (специальное значение <i>non-value</i> – нет значения)	<code>a or b</code>	Если <code>a</code> равно 0 или None, использовать <code>b</code> , иначе использовать <code>a</code>

Термин *тождественная* или *не тождественная*, использованный в предыдущей таблице, означает, что переменные содержат (не содержат) ту же самую информацию. Например, выражение `a is b` возвратит `true`, если переменным `a` и `b` присвоить значение `hello` или обеим этим переменным присвоить значение `1`.

В этой главе Вы также познакомились с основными инструкциями управления логикой выполнения программы: `for`, `while`, `break`, `continue` и `pass`. ещё Вы узнали о функции `range()`, которая часто используется с инструкцией `for`. И в заключение мы создали работоспособную программу, в которой использовали средства программирования, рассмотренные на этом занятии.

Практикум

Вопросы и ответы

Сколько всего существует инструкций управления логикой выполнения программы?

Немного, всего только шесть вместе с инструкцией `goto`, которая, к превеликому счастью, отсутствует в Python. Но в действительности их вполне достаточно для выполнения всех возможных задач.

Для чего нужны эти знаки двоеточия в конце строк?

Двоеточия используются для явного обозначения окончания выражений с инструкциями, такими как `if` и др. Хотя вполне приемлемо использовать в коде программы инструкцию `if`, например, в следующем виде:

```
if a<b<c: print a, b, c
else: print c, b, a
```

В этом случае использование двоеточия является единственным способом отделить собственно инструкцию `if` от сопутствующего ей блока выражений. Но с моей точки зрения, стиль, который я использовал до сих пор (и буду использовать в дальнейшем), более удобен для понимания, хотя, безусловно, это дело вкуса.

Очевидно, что использование имени функции в качестве имени переменной трудно назвать хорошей практикой программирования, поскольку это чревато ошибками в программах на языке Python.

Существуют ли языки, которые были бы настолько сообразительны, чтобы отличать имена встроенных функций от одноимённых переменных? (Например, позволяли бы использовать `int` или `range` в качестве имени переменной.)

Трудно сказать, следует ли такую возможность отнести к сообразительности или к недостатку языка. В любом случае такое использование имён никогда не станет хорошим стилем программирования.

Чем так плоха инструкция `goto`? Разве способность продолжать выполнение программы с любой строки не расширила бы возможности программиста?

Проблема с инструкцией `goto` большей частью состоит не в возможности переходить к выполнению любой строки программы, а в том, чтобы отследить, откуда Вы пришли и как Вы сюда попали, что чревато большой путаницей.

Контрольные вопросы

1. Что такое переменные?
 - а) Иксы и игреки.
 - б) Числа, которые беспорядочно изменяются, когда Вы за ними не следите.
 - в) Имена, символизирующие конкретные значения, которые можно изменять.
 - г) Звезды с изменяющейся яркостью свечения.
2. Каким термином называют блок выражений, следующий за инструкциями `if`, `for` и `while`?
 - а) Тело.

- б) Фрагмент.
- в) Фрагмент тела.
- г) Массив.

3. Что означает выражение `a>b<c`?

- а) Это недопустимая комбинация операторов, которая в Python не имеет никакого смысла.
- б) Объединить значения переменных `a` и `c` и поместить результат в переменную `b`.
- в) Сравнить указанные три значения и, если `b` меньше остальных двух, вернуть `true`.
- г) Вывести на печать переменную `b` только в случае, если `a` и `c` больше её.

Ответы

- 1. в. Переменные – это имена, символизирующие конкретные значения, которые можно изменять.
- 2. а. Правильный термин для обозначения блока выражений, следующего за инструкциями `if`, `for` и `while`, – тело. Выражения, относящиеся к одному блоку, выделены в коде одинаковым отступом.
- 3. в. Данное выражение сравнивает значения переменных и, если переменная `b` меньше двух остальных, возвращает значение `true`. Более того, Вы не ограничены только двумя операторами сравнения. Вы можете записать строку `if a<b<c<d< ...` и продолжать цепочку сравнения сколько угодно долго

Примеры и задания

Попробуйте заново составить программу вычисления високосного года, представленную в листинге 4.1, чтобы в ней использовалась инструкция `while` вместо `for`. Попробуйте, по мере сил, усовершенствовать код программы.

Найдите в Internet ссылки на самые первые компьютеры и посмотрите, имеется ли какая-нибудь информация о языках программирования для них. Посмотрите также, какие инструкции управления логикой выполнения программ имелись в этих первых языках программирования.

5-й час

Основные типы данных; числовые

Изучая материал предыдущих четырех глав, мы подготовили почву для дальнейшего изучения Python – познакомились с

базовыми концепциями объектно-ориентированного программирования и узнали о переменных. Но что мы знаем о том, что может храниться в этих переменных? Вы, вероятно, уже поняли, что переменные могут содержать самую различную полезную информацию, включая значения дат и времени, строки символов и даже целые функции. Но при этом нужно следовать определённым правилам сохранения данных и их интерпретации в Python. Чтобы разобраться в этих правилах, следует иметь представление о типах данных, поддерживаемых в Python. Этой теме как раз и посвящены настоящая и следующая главы. Здесь мы обсудим типы данных, представляющих числовые значения:

- целое (integer);
- длинное целое (long integer);
- число с плавающей запятой (floating point);
- комплексное число (complex).

Типы данных

Для компьютера наступают трудные времена, когда он не может понять, что именно подразумевал программист в своей программе под той или иной переменной. Компьютеру необходимо чётко и ясно знать, сколько места в памяти следует отвести для данной переменной и как обрабатывать её содержимое. Отнесение переменной к определённому типу данных является тем методом, с помощью которого в программе можно явно указать все эти характеристики. *Типы данных* — это категории, на которые можно подразделить единицы информации, обрабатываемые программой. Типы данных в Python можно объединить в следующие категории более высокого порядка: числовые данные, последовательности, функции и типы, определяемые пользователем. Типы, определяемые пользователем, будут рассмотрены в части III этой книги. Числовые типы рассматриваются в этой главе, а остальные — в следующей.

Типы данных в Python (как и в большинстве других языков программирования) делятся на две общие категории ещё более высокого порядка: основные (или базовые) и пользовательские (определяемые пользователем). Ко второй категории относятся объекты, с которыми, как уже говорилось, мы познакомимся ближе несколько позже. Объекты — это важнейшая составная часть объектно-ориентированного программирования, хотя различия между объектами и основными типами данных не всегда очевидны, особенно в случае применения функций. Кроме того, не все объекты определяются пользователем.

В этой и следующей главах мы будем оперировать только с основными типами данных, к которым относятся *числовой*,

последовательность и словарь. Функции будут рассмотрены в главе 7, остальные пользовательские типы, или объекты, освещаются в части II. А сейчас займёмся числовыми типами данных.

Числовые типы

Компьютеры не наделены собственным интеллектом, поэтому программистам в большинстве языков программирования необходимо явно указывать, к какой разновидности чисел относится то или иное значение. Более того, компилируемым языкам эту информацию надо сообщать заранее, т.е. тип переменной должен быть объявлен до инициализации переменной. (Инициализация — это присвоение переменной первого значения.) Попытка присвоить переменной значение, не соответствующее объявленному для неё типу, закончится ошибкой в работе компилятора (чем он Вас непременно "порадует"). Например, предположим, что в C Вы объявляете переменную следующим образом:

```
int n;
```

Эта строка подразумевает, что переменная *n* будет содержать положительные или отрицательные целые числа. Предположим, что где-то в коде своей программы Вы попытаетесь выполнить такое присвоение: `n=3.14159;`

Компилятор языка C выразит Вам своё недовольство сообщением об ошибке и не позволит выполнить эту операцию. В отличие от C, Python с готовностью позволит Вам на ходу изменить тип переменной, причём без единой жалобы. Это объясняется способностью Python динамически определять типы переменных, в то время как в C переменные являются статичными. Другими словами, после объявления в C типа переменной компилятор строго следит за соответствием присваиваемых значений этому типу и не позволяет менять тип во время выполнения программы. В Python же тип переменной определяется в момент присвоения ей значения, поэтому тип переменной можно изменять столько, сколько вашей душе будет угодно. Профессиональный программист скажет, что такое свойство языка относится скорее к его недостаткам, чем к достоинствам. Очень трудно разобраться в коде, если одна и та же переменная в одном блоке содержит дату, а во втором — текст. Python предоставляет программистам множество возможностей, но старайтесь использовать эти возможности на пользу, а не во вред своей программе.

***Прим. В. Шипкова: последний совет, по существу, довольно**

полезен. Мало того, что результат может оказаться ошибочным в плане точности вычислений, ещё и сами по себе преобразования из одного типа в другой это довольно затратная операция для процессора. Не стоит об этом забывать.

Но если изменения типа переменной в ходе выполнения программы — это не лучшая идея, то почему же тогда такая возможность допускается? В каких случаях выгодно динамическое определение типов? Почему бы не заставить программистов объявлять заранее тип переменной, а затем заставить их строго придерживаться собственных объявлений? В конце концов, это ведь именно то, что делает большинство других современных языков программирования.

Работа Python базируется на предположении, что программист знает, что делает. Статичность типов переменных в других языках — это результат реализации противоположной концепции, заключающейся в том, что программист нуждается в защите непосредственно от самого себя. Динамическое определение типов, если им не злоупотреблять, позволит существенно сократить и упростить код программы. Например, предположим, что Вы считываете строки символов, вводимые пользователем. Вы хотите преобразовать эти строковые данные в числа и использовать их в качестве входных данных своей программы. Если пользователю разрешается вводить целые числа, длинные целые и числа с плавающей запятой, то на обычном языке программирования Вам пришлось бы для этого предусмотреть по крайней мере три переменные трёх разных типов. А в Python Вам хватит всего лишь одной переменной, которая может содержать любое значение из трёх допустимых типов. И это становится возможным благодаря динамическому определению типов данных в Python.

Мы начнём подробное обсуждение темы с уже знакомого Вам типа — *целые числа*.

Целые числа

Тип *целые числа* (*integer*), как следует из его названия, представляет собой весь числовой ряд положительных и отрицательных целых чисел, а также нуль. Как и в других языках программирования, смысл использования данного типа данных состоит в экономии памяти, так как для сохранения целочисленного значения требуется меньше места, чем для значений других числовых типов. Большинство языков, включая Python, реализуют целые числа в виде 32-битовых значений, что ограничивает диапазон допустимых целочисленных значений пределами от `-maxint` до `+maxint`. Чтобы оценить допустимый

диапазон значений на Вашем компьютере, запустите Python в режиме командной строки или воспользуйтесь редактором IDLE, импортируйте модуль `sys` и попробуйте выполнить ряд операций с числами `maxint`, как показано в листинге 5.1. На моём компьютере с Linux получились следующие значения (строки, вводимые пользователем, выделены полужирным шрифтом).

Листинг 5.1. Операции с `maxint`

```
Python 1.5.2 (10, Apr 13 1999, 10:51:12)
[MSC 32 bit (Intel)] on Win32
Copyright 1991-1995 Stichting Mathematisch Centrum,
Amsterdam
>>> from sys import *
>>> maxint
2147483647
>>> -maxint
-2147483647
>>> maxint + 1
Traceback (innermost last):
File "<pyshell#3>", line 1, in ?
maxint + 1
OverflowError:
integer addition
>>>
```

В некоторых системах целые числа представляются 64-битовыми значениями, а в системах Cray, насколько я знаю, для целых чисел отводится 128 бит. Тем не менее сообщение об ошибке `OverflowError`, которое Вы видите в приведенном выше примере, все равно появилось бы. Просто диапазон чисел, лежащих между значениями `-maxint` и `+maxint`, был бы намного больше. Я никогда не работал на столь мощных машинах, но с помощью другого типа данных Python — `long integer` (длинное целое) — можно оценить диапазоны целых чисел у таких машин. В листинге 5.2 приведён код программы, который необходимо ввести, сохранить в файле `inttest.py` и запустить с помощью Python. Кстати, что касается всех листингов кода, представленных в этой книге, то вместо того чтобы набирать их вручную, можно просто выгрузить их со страницы этой книги в Internet.

```
from sys import *
z = long ( maxint )
print "32-bit machine:"
print "maxint:",z,"-maxint:",-z,"(2**31L)-1:",(2**31L)-1
print "-----"
y = (2 ** 63L) - 1
```

```
print "64-bit machine:"
print "maxint:", y, "-maxint:", -y
print "-----"
z = (2 ** 127L) - 1
print "128-bit machine:"
print "maxint:", z, "-maxint:", -z
```

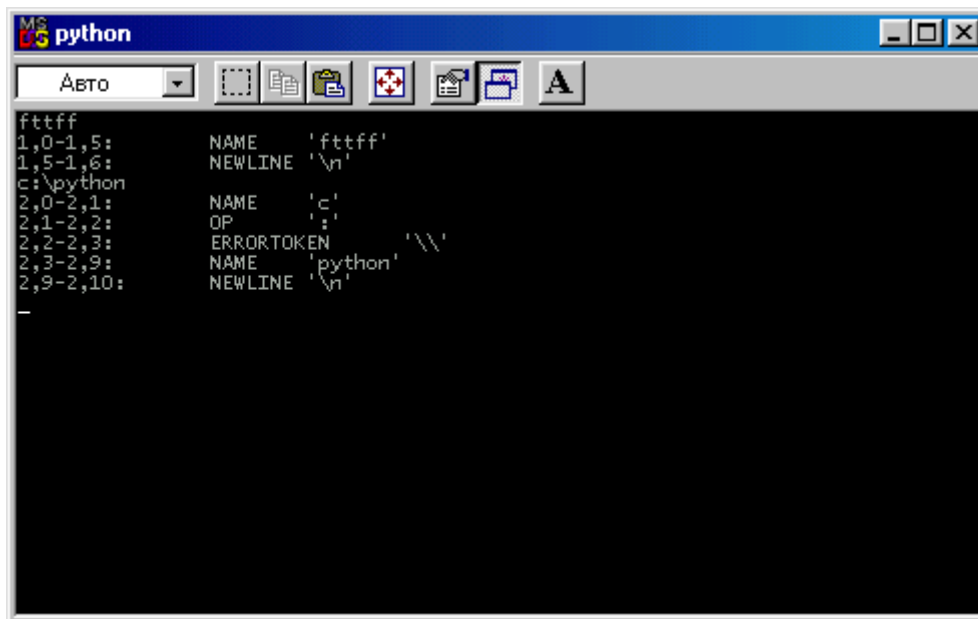
На рис. 5.1 показан результат выполнения программы `inttest.py`.

Обратите внимание, что на 32-разрядной машине самым большим целым числом, которое можно использовать, является значение $(2^{31}L) - 1$. Это объясняется тем, что один бит используется для указания знака числа — положительного или отрицательного. В большинстве систем он называется *старшим двоичным битом*, или просто *знаковым битом*. Причина того, что Вы не можете использовать число, которое точно равно значению $2^{31}L$ (т.е. 2147483648), довольно сложна и запутанна, чтобы объяснять её в данный момент. Она связана со способом представления чисел в двоичной системе.

**Прим. В. Шипкова: я не считаю, что это столь уж сложно и непонятно. Поэтому объясняю. Нумерация ячеек памяти в компьютерах начинается с 0. Т. е. 1-ая ячейка памяти имеет адрес 0, 2-ая — 1 и т. д. Таким образом, как адресация, так и численное значение ячеек памяти будут иметь численный предел меньший на единицу, чем можно было ожидать. Почему так? Да потому что, когда на шину адреса выставляются нули — это тоже число, и оно тоже должно что-то адресовать. Именно по этому, число ноль, частенько относят к положительным. ИМХО, это глюк разработчиков — по моему, когда на шину адресов выставляется ноль, это должно быть эквивалентно высокоимпедансному состоянию. Это когда микросхемы принудительно отключаются от шин. Сразу оговорюсь, при текущих схемах адресации пришлось бы пожертвовать нулевым байтом, т. к. он бы стал недоступен. С другой стороны, если хорошо подумать, то по предложенному мной варианту схема адресации была бы вряд ли проще (а на сколько сложнее — надо полагать, уж посложней). Вот и мучаются программисты с системщиками забывая про эту "мелочь".*

Заинтересованным читателям рекомендую обратиться к разделу "Примеры и упражнения" этой главы, где помещены ссылки на дополнительные материалы. В их числе Вы найдёте серверы, посвященные исследованиям позиционных числовых систем, где параллельно освещаются и вопросы компьютерного представления чисел. Другие компьютерные языки позволяют

хранить целые числа без знака, и такие числа, что очевидно, называются **беззнаковыми** (unsigned). Беззнаковые числа позволяют использовать все 32 бита, но только для представления положительных целых чисел.



```
python
Авто
fttff
1,0-1,5:      NAME      'fttff'
1,5-1,6:      NEWLINE  '\n'
c:\python
2,0-2,1:      NAME      'c'
2,1-2,2:      OP        ':'
2,2-2,3:      ERRORTOKEN  '\\'
2,3-2,9:      NAME      'python'
2,9-2,10:     NEWLINE  '\n'
```

Рис. 5.1. Результат выполнения программы *inttest.py*

Аналогично, для 64-разрядной машины самым большим целым числом, которое допускается использовать, является значение $(2^{63}) - 1$, а для 128-разрядной машины оно составляет $(2^{127}) - 1$. В своей работе Вы, как правило, не будете достигать граничного значения `maxint` даже на 32-разрядных машинах. Максимальные значения целых чисел в 64- и 128-разрядном представлении — это очень большие числа, которые могут понадобиться только для космических исследований или подсчетов числа молекул.

Но если проблема с максимально (минимально) допустимыми значениями у Вас всё же возникнет, не отчаивайтесь. Длинные целые числа — вот способ, позволяющий обойти эти ограничения в Python, и мы обсудим их в следующем разделе.

Длинные целые числа

Длинные целые числа типа *long integer*, которые иногда встречаются также под названиями *longs* или *bigints*, присутствовали в Python с самого начала. По существу, они ведут себя точно так же, как и обычные целые числа, но только не ограничены никакими пределами. Преобразование целых чисел в длинные целые числа и наоборот, к сожалению, не является достаточно прозрачной процедурой для программистов. Но есть надежда, что Guido устранил эти различия в будущих версиях Python. Впрочем,

можно попробовать самостоятельно составить алгоритм, который будет автоматически выполнять эти преобразования, если Вам это потребуется.

Что подразумевается под словами *не ограниченные никакими пределами*? В действительности это совсем не означает, что Вы можете использовать числа, уходящие в бесконечность. Имеется в виду только то, что обычные ограничения существенно расширяются. К чему это сводится на практике? К тому, что размер любого заданного длинного целого числа подчиняется только фактическим ограничениям, вытекающим из характеристик вашей машины. Например, Вы не можете использовать в своей программе такое число, для хранения которого потребуется больший объём памяти, чем её имеется фактически. Так, на моей домашней машине с системой Server Windows NT 4.0 установлено 256 Мбайт оперативной памяти. В идеале, если пренебречь обязательным расходом памяти на функционирование операционной системы и самого интерпретатора Python, а также ряда обязательных утилит, самое большое длинное целое число, которое я мог бы теоретически использовать, должно уместиться в эти 256 Мбайт. В результате получилось бы число длиной что-то около $2^{013\,265\,920}$ цифр. Это довольно приличное значение, но, тем не менее, конечное. Во внутреннем компьютерном представлении длинные целые хранятся как 16-разрядные числа с максимальным значением 32 768. Поскольку 32 768 — это 2^{15} — это 2 в 15-й степени, становится очевидным, что 16-й разряд используется для чего-то ещё. Этот 16-й бит применяется для реализации "флага переполнения" (т.е. выхода за пределы), который используется для организации операций по переносу, заимствованию и т.п., что во многом напоминает то, как Вы поступаете, когда решаете расчетные задачи на бумаге. Чтобы лучше уяснить то, как устроены большие целые числа, попробуйте выполнить упражнения, приведенные в конце этой главы.

Во время работы с длинными числами внимание следует обращать не только на возможности Вашего компьютера, но и на терпение пользователей вашей программы. Например, если в вашей программе предполагается вывод на печать числа гуголплекс, то пользователю придётся посидеть у своего компьютера лет 5, а то и больше.

С другой стороны, длинные целые числа просто необходимы для выполнения некоторых вычислений. Предположим, что Вам нужно преобразовать расстояние в световых годах в километры. С помощью Python это вычисление выполняется очень просто. Вам достаточно только знать, что скорость света (c) приблизительно равна 300 000 км/с. (Более точно



значение, а также принципы расчета скорости света представлены на Web-странице по адресу <http://ww.ph.uniraelb.edu.au/~nibailes/P140/lecture22/index.htm>.) В листинге 5.3 приведена простая процедура определения длины космического светового года в километрах.

```
def c(p) :  
    spy = 60*60*24*365.2422  
    n = long(spy)*long(p)  
    return n  
  
if __name__ == "__main__":  
    n = c(300000)  
    print n
```

Хотя Вы ещё не знакомы с функциями, речь о которых пойдет только в следующей главе, я позволил себе забежать немного вперед и реализовал эту процедуру как вызов функции, в которую в качестве аргумента передается скорость света в вакууме. Если Вас не удовлетворяет точность вычислений и Вы захотите использовать более точное значение скорости света, достаточно просто изменить аргумент функции `c()` в строке 6.

Число 365,2422 в листинге 5.3 – это приблизительная длина года, выраженная в днях. Для расчетов дат по григорианскому календарю используется значение длины года 365,2425 дней. (В юлианском календаре использовалось менее точное приближение – 365,25 дней.) Тем не менее, к вашему сведению, астрономы, оперирующие световыми годами, используют в расчетах юлианский год, поскольку в действительности на расстояние, пробегаемое световой волной за год, влияет так много факторов, что большая точность вычислений будет статистически недостоверной. Астроном Дункан Стил (Duncan Steel) написал короткую заметку, в которой излагает несколько серьезных доводов, обосновывающих предпочтительность использования в вычислениях юлианской длины года. С этой статьей можно познакомиться на Web-странице по адресу <http://www.pauahtun.org/TYPython/steel.html>.

Запустив на выполнение короткую программу `c.py`, получим следующий результат: 9467077800000L

Это, действительно, более 9×10^{12} км. Так нам говорили ещё в школе, хотя это расстояние трудно себе представить. Использование более точного значения скорости света в вакууме позволит повысить точность вычисления, но порядок и невообразимость конечного результата останутся теми же.

Числа с плавающей запятой

В предыдущих главах мы уже встречались с числами с плавающей записей и с экспоненциальным представлением чисел. Надеюсь, Вы уже достаточно чётко представляете, о чем идёт речь. Числа с плавающей запятой (*floating point*) в Python являются аналогом типа *double* (числа с двойной точностью) в языке C. Двойная точность подразумевает выделение для их хранения 64 бит памяти. Часть этих битов выполняет служебные функции, поэтому фактически для данного типа самым большим и самым маленьким допустимыми значениями являются (приблизительно) $1.79769e+308$ и $2.22507e-308$ соответственно, по крайней мере на большинстве компьютеров. Плохо то, что Python не даёт вразумительного сообщения об ошибке в случае выхода значения с плавающей запятой за допустимые пределы. Вместо этого получаются довольно странные результаты. Посмотрите, что получится, если назначить переменной максимальное значение числа с плавающей запятой, а именно $1.79769e+308$, а затем удвоить его. На своей машине с NT я получил следующий результат: 1. * INF. К счастью, существуют способы заставить Python сообщать Вам о возникновении ошибок подобного рода. Мы рассмотрим их в последующих главах.

***Прим. В. Шипкова: я не поленился проверить, как обстоят с этим дела в версии 2.4. Что же я получил? 1.#INF
Так что, поосторожней с этим делом.
Я думаю, это не связано конкретно с Питоном. Скорей всего - это результат работы сопроцессора.**

На тех машинах, где допускается резервирование 64 и 128 битов для целых чисел, как правило, для чисел с плавающей запятой также выделяется больше разрядов, что расширяет допустимый диапазон значений. Следует отметить, что в Python выделение памяти для значений основных типов данных происходит в соответствии с особенностями аппаратных средств. Поэтому по мере увеличения производительности машины более мощным становится и сам Python.

Если Вас интересуют характеристики собственного компьютера, загрузите из Internet файл <http://www.pauahtun.org/TYPython/machar.zip>. Это программа на языке C, которая после запуска на выполнение сообщает максимальные и минимальные предельные значения чисел с плавающей запятой. Для систем Linux и DOS программа доступна как исходный код программы, так и выполняемый файл. Для других систем программу нужно скомпилировать самостоятельно.

Комплексные числа

Комплексные числа, как и длинные целые, были и остаются составной частью Python. Не так уж много найдется языков, которые обеспечивают встроенную поддержку использования комплексных чисел. ещё один язык, который может похвастать этой особенностью, — это FORTRAN. Но данное свойство принесет ощутимую помощь только в том случае, если Вы знаете, что представляют собой комплексные числа.

Комплексные числа успешно прижились в электронике, где благодаря их применению с восхитительной простотой описывается поведение сложных электроцепей. Они также весьма эффективны в описании механики небесных тел, в частности для решения систем уравнений, которые помогают астрономам рассчитывать орбиты планет, спутников и астероидов. А не так давно они нашли широкое применение в компьютерной графике, во многом благодаря их популяризации (если допустимо употребить здесь это слово) Беноитом Мандельбротом (Benoit Mandelbrot), сотрудником корпорации IBM, который открыл математическое множество, названное теперь его именем. В последующих главах Вы увидите, как можно использовать комплексные числа для рисования наборов Мандельброта (Mandelbrot set).

Чтобы уяснить, что представляют собой комплексные числа, вспомните материал предыдущих глав, в которых целые числа мы рассматривали в виде значений, расположенных на одной линии с нулем в центре, протяжённой бесконечно в обоих направлениях, — отрицательном и положительном. (Хотя в действительности в компьютере они не уходят в бесконечность, а ограничены минимальным и максимальным допустимыми значениями.) Вот пример целых чисел: $\dots -3, -2, -1, 0, 1, 2, 3 \dots$. Длинные целые числа в Python находятся на той же самой числовой оси, только их пределы дальше отдалены влево и вправо. На этой же числовой оси располагаются и числа с плавающей запятой, только они заполняют ось сплошным рядом. Если каждый отрезок числовой оси, расположенный между двумя смежными целыми числами, разбить на бесконечное множество точек, то это и будут числа с плавающей запятой. Совокупность целых чисел и чисел с плавающей запятой называется *вещественные числа*.

Когда мы говорим о числах и числовой оси, то фактически подразумеваем элементы четырех числовых систем.

1. Натуральные, или счётные, числа: 1, 2, 3 и т.д.
2. Целые числа: $-3, -2, -1, 0, 1, 2, 3$ и т.д.
3. Рациональные числа: $3/4, 5/8, 1/10$ и т.д.

4. Вещественные числа: 3,14159, 1,001 и т.д., включая иррациональные числа.

Возможно, Вы уже слышали об иррациональных числах, среди которых число "пи" является самым известным (отношение диаметра к радиусу окружности). Эти числа могут быть выражены только как отношение двух других чисел (в виде дроби a/b). Если попытаться определить их фактическое значение, то мало того, что они не только не дают никакого конечного значения, типа $1,5/0,5 = 3,0$, но не получится даже периодической десятичной дроби, такой как $10,0/3,0 = 3,33333333333333...$. Вместо этого получается бесконечная дробь без какого бы то ни было намёка на подобие закономерности. В научно-фантастическом рассказе "Контакт" писателя Карла Сагана (Carl Sagan — "Contact") описывается сообщение, скрытое в числе "пи" в районе десятичных разрядов, расположенных где-то за сотней миллиардов знаков после запятой. Если бы это было правдой, мы действительно имели бы послание богов, а не только каких-то там инопланетян. Но любое сообщение — это закономерность, а чередование знаков в числе π — это случайность. Так что поиск подобных посланий противоречит фактам современной математики.

Несмотря на то что данные числа называются иррациональными, они все равно расположены на числовой линии. И, как и в случае с любыми другими числами, количество иррациональных чисел на числовой прямой бесконечно. Числа, которые не являются иррациональными, вполне естественно, называются рациональными.

Существует, однако, ещё одна категория чисел, которые вообще невозможно расположить на числовой прямой. Это — мнимые числа. На ранних этапах развития математической науки даже отрицательные числа рассматривались как бессмысленные. Имеются в виду математические представления древних цивилизаций, таких как индейцы Майя. Чтобы представить себе отрицательные числа, нужно было обладать достаточным уровнем абстрактного мышления. Позже, когда появились первые намёки на то, что теперь называется мнимыми числами, математики по инерции были склонны скорее игнорировать собственные же открытия.

Классический пример мнимого числа — квадратный корень из -1 . Квадрат некоторого числа по определению является результатом умножения этого числа самого на себя. Тогда само это число называется корнем. Например, 5 является квадратным корнем числа 25. ещё со школы мы помним, что при умножении двух чисел с одинаковым знаком, даже если этот

знак отрицательный, результат всегда будет положительным. Следовательно, знак подкоренного числа обязательно должен был положительным. Невозможно извлечь квадратный корень из отрицательного числа. Но тут возникает противоречие. Поскольку отрицательные числа все же существуют и их можно возводить в квадрат, то должна быть возможность и извлекать из них корень. Перед нами возникает та же проблема отрицательных значений, которая стояла перед древними математиками. Очевидно, что они должны существовать, но их невозможно себе представить.

Чтобы использовать мнимые числа, их следует представить в формате комплексных чисел. Современные математики помещают мнимые числа на оси, перпендикулярной числовой оси вещественных чисел. Тогда комплексные числа представляются в виде совокупности вещественного и мнимого компонентов. Комплексные числа подчиняются правилам сложения, вычитания и умножения. Фактически любое действие, которое может быть выполнено над вещественными числами, может быть также выполнено и над комплексными числами. Следуя этим правилам, мы можем вычислить квадратный корень из -1 . Вещественная часть комплексного числа пишется просто как она есть, а мнимая составляющая представлена числом, за которым следует суффикс — буква i или j . Поскольку в Python используется буква j , я также буду применять её в этой книге. Квадратный корень числа -1 записывается в виде $(0, 1j)$. Умножение комплексных чисел выполняется в соответствии со следующим правилом: $(a,b)(c,d) = (ac-bd, ad+bc)$

Выполним его для числа $(0, 1j)$ и получим следующий результат:

```
(0, 1j)(0, 1j)
{0*0) - (1*1), (0*1) + (1*0)
(0-1), (0+0)
(-1, 0j)
```

Как видим, при отображении результата на числовой плоскости мнимый элемент $(0, 1j)$ пропадает, в результате чего получаем вещественное число -1 .

Подобные действия просто выполнять в Python, потому что в него встроена поддержка комплексных чисел. Все, что необходимо сделать в Python, — это определить число как комплексное. В этом нет никаких проблем. Следующий пример демонстрирует, как сообщить Python, что переменная a является комплексным числом:

```
a=0+1j
```

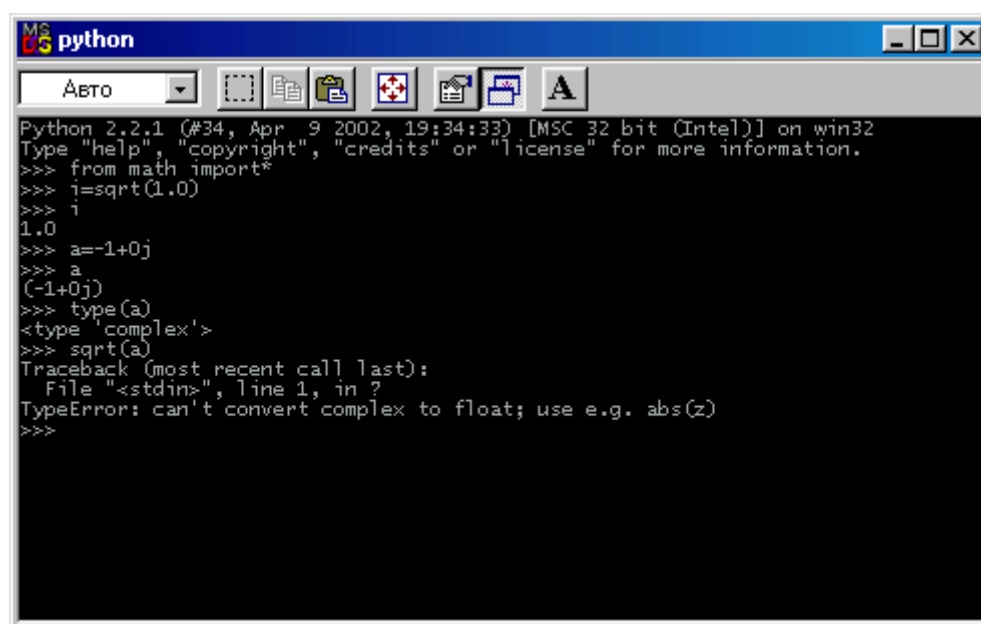
Возвести в квадрат это комплексное число также просто:
`s=a*a`

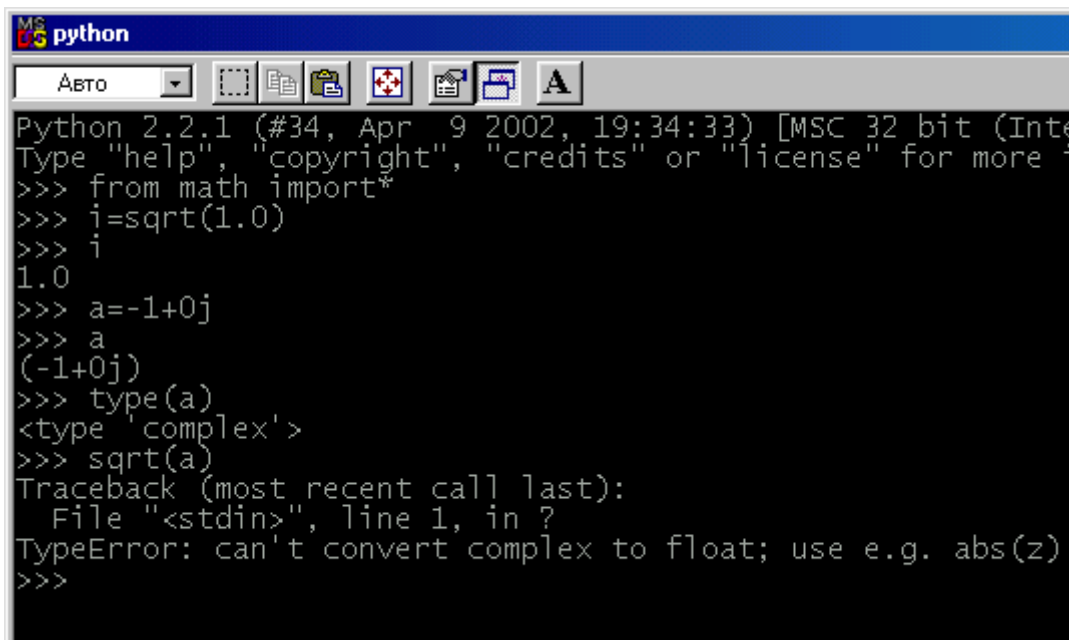
Python зорко отследит комплексный характер переменной `a` и автоматически сделает комплексным результат возведения этого числа в квадрат (переменная `s`). Выполнение обычных математических операторов над комплексными числами будет давать правильный результат, в чём можно убедиться, если перемножить два комплексных числа. Фактически любое действие, которое можно выполнять над числами с плавающей запятой, может быть применено для комплексных чисел. Правда, для этого нужно импортировать в программу модуль математических функций для обработки комплексных чисел. Вспомните, что в предыдущих главах для получения доступа к математическим функциям, таким как `sqrt()`, приходилось импортировать модуль `math`: `from math import *`

Эта строка указывает Python, что необходимо найти математический модуль `math` и добавить в код программы все определения функции, которые там есть. В результате Вы получаете возможность использовать эти функции в своей программе, как, например, в следующем примере:

```
i = sqrt (1.0)
```

Если мы попытаемся выполнить ту же операцию с комплексными числами, то Python выразит своё недовольство, как показано на рис. 5.2.





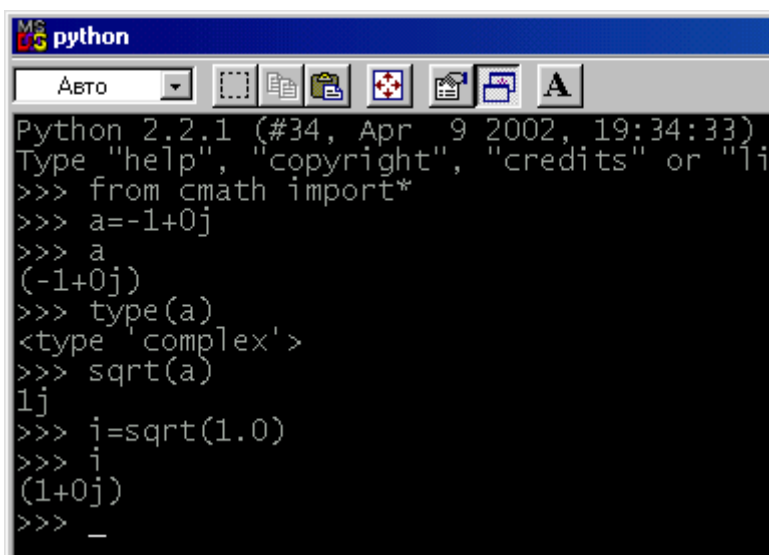
```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)]
Type "help", "copyright", "credits" or "license()" for more
>>> from math import *
>>> i=sqrt(1.0)
>>> i
1.0
>>> a=-1+0j
>>> a
(-1+0j)
>>> type(a)
<type 'complex'>
>>> sqrt(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>>
```

Рис. 5.2. Функции модуля *math*

Чтобы использовать математические функции для обработки комплексных чисел, следует импортировать специальный модуль *cmath*. Синтаксис импортирования модуля остаётся тот же:

```
from cmath import *
```

Повторим опять нашу операцию с комплексным числом. В этот раз Python не покажет сообщения об ошибке, а возвратит правильный результат (рис. 5.3).

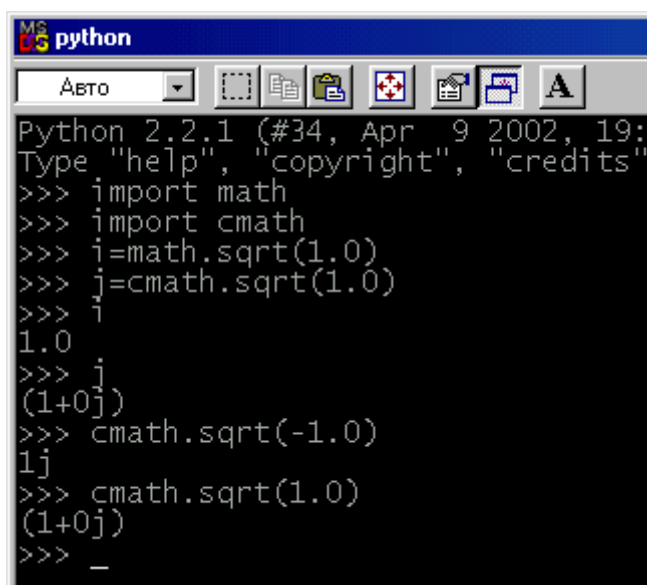


```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33)
Type "help", "copyright", "credits" or "license()" for more
>>> from cmath import *
>>> a=-1+0j
>>> a
(-1+0j)
>>> type(a)
<type 'complex'>
>>> sqrt(a)
1j
>>> i=sqrt(1.0)
>>> i
(1+0j)
>>> _
```

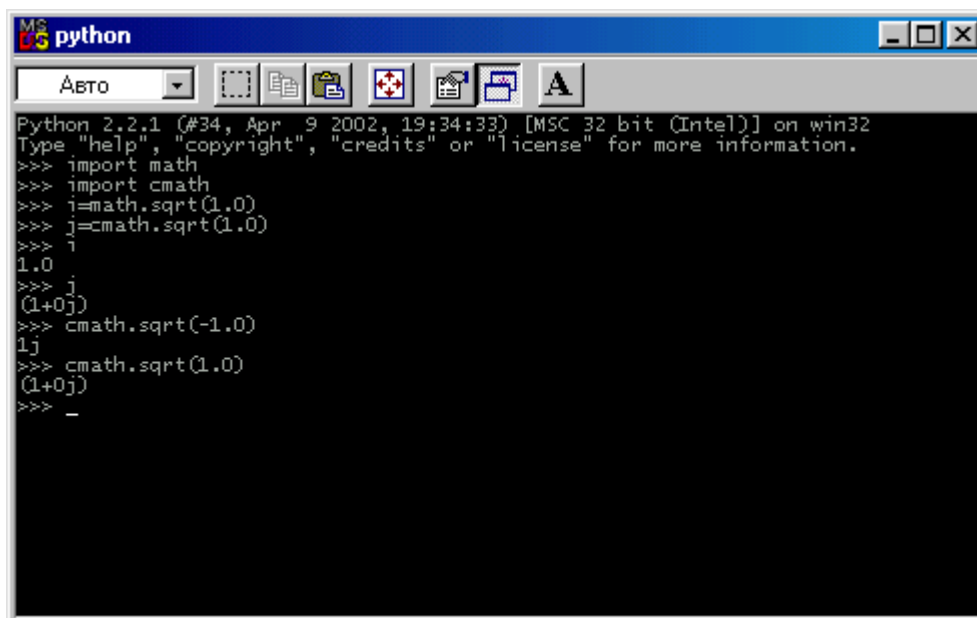
Рис. 5.3. Модуль математических функций для комплексных чисел

Но может возникнуть другая проблема. Внимательно рассмотрите введенные строки на рис. 5.3. Все функции в модулях *math* и *cmath* имеют одинаковые имена. Если последним был импортирован модуль *cmath*, то все функции чисел с

плавающей запятой заменяются их эквивалентами для комплексных чисел. В результате все числа с плавающей запятой будут обрабатываться как комплексные. Но что делать, если необходимо использовать функции из обоих модулей? Этот случай отчётливо подчеркивает одну из опасностей импортирования модулей выражением `from module import *`. Рано или поздно Вы обязательно потеряете что-нибудь из того, что Вам необходимо! Решение этой проблемы заключается в том, чтобы использовать идентификаторы, явно указывающие модуль-источник функции. Пример использования идентификаторов показан на рис. 5.4.



```
Python 2.2.1 (#34, Apr 9 2002, 19:
Type "help", "copyright", "credits"
>>> import math
>>> import cmath
>>> i=math.sqrt(1.0)
>>> j=cmath.sqrt(1.0)
>>> i
1.0
>>> j
(1+0j)
>>> cmath.sqrt(-1.0)
1j
>>> cmath.sqrt(1.0)
(1+0j)
>>> _
```



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> import cmath
>>> i=math.sqrt(1.0)
>>> j=cmath.sqrt(1.0)
>>> i
1.0
>>> j
(1+0j)
>>> cmath.sqrt(-1.0)
1j
>>> cmath.sqrt(1.0)
(1+0j)
>>> _
```

Рис. 5.4. Использование идентификаторов модулей функций для комплексных чисел и для чисел с плавающей запятой

Символ `(.)` является оператором прямого обращения. Он используется для указания модуля, из которого необходимо

взять данную функцию. Например, на рис. 5.4 мы попеременно обращались к одноименным функциям `sqrt()` двух модулей – `math` и `cmath`. Оператор прямого обращения `(.)` используется также для получения доступа к членам классов, о чем мы узнаем в разделе, посвященном объектам.

Резюме

В данной главе мы обсудили типы данных вообще и числовые типы в частности. Вы познакомились с такими типами, как целые числа, длинные целые числа, числа с плавающей запятой и комплексные числа, а также с подходами их реализации в Python. Вы узнали о комплексных числах и их отношении с вещественными числами, а также об имеющихся функциях для их комплексных чисел. Вы узнали также о том, какие вообще существуют числовые системы, и научились правильно импортировать модули с одноименными функциями. В следующей главе мы рассмотрим оставшиеся типы данных: последовательности и словари.

Практикум

Вопросы и ответы

Где в реальном мире используются комплексные числа?

Сила электромагнитного поля – вот наиболее яркий пример. Поскольку такое поле имеет электрический и магнитный компоненты, то в качестве единицы измерения этой силы можно использовать комплексное число.

Я выполнил на своём компьютер программу `machar.exe` и определил, что для представления чисел с плавающей запятой используется только 53 бита из 64. Чем это можно объяснить?

В действительности 53 бита используются только для мантиссы, т.е. дробной части, а ещё 11 битов используются для показателя степени. В сумме для представления значения с плавающей запятой используется 64 бита.

Контрольные вопросы

1. Из каких двух компонентов состоят комплексные числа?
 - а) Вещественного и не вещественного чисел.
 - б) Мнимого и кошмарного.
 - в) Мнимого и вещественного.
 - г) Ужасного и прекрасного.

2. Какие пять числовых систем упоминались в этой главе?
- а) Натуральные, искусственные, мнимые, рациональные и иррациональные.
 - б) Натуральные, целые, рациональные, вещественные и комплексные.
 - в) Ордовикские, силурийские, девонские, юрские и триасовые.
 - г) А, В, С, D и Е.
3. Что произойдет, если импортировать два модуля, в которых имеются одноимённые функции?
- а) В случае использования идентификаторов не произойдет ничего страшного.
 - б) Если использовать инструкцию `import *`, функции второго модуля заместят одноимённые функции первого.
 - в) Одноименные функции обоих модулей окажутся недоступными.
 - г) Окажутся недоступными все функции обоих модулей.

Ответы

1. в. Комплексные числа состоят из мнимой и вещественной составляющих.
2. б. Пять числовых систем: натуральные, целые, рациональные, вещественные и комплексные числа.
3. а и б. Если импорт модулей выполняется с помощью ключевого слова `import` без звездочки, то перед именами функций следует указывать имена модулей, что одновременно предотвратит возникновение конфликтов имён. Если импортировать модули с помощью инструкции `import *`, то функции первого модуля будут замещены одноименными функциями второго модуля. Идентификаторы в этом случае не исправят положения.

Примеры и задания

Для ознакомления с вводным материалом по мнимым, комплексным, рациональным и иррациональным числам, а также по другим близким темам загляните на Web-узел по адресу <http://www.math.toronto.edu/mathnet/answers/imaginarily.html>.

Интересные сведения о календаре индейцев Майя представлены на Web-странице по адресу <http://www.foretec.com/Python/workshops/1998-11/proceedings.html>. Здесь же Вы узнаете о том, как представляли себе отрицательные числа мудрецы Майя.

Просмотрите ещё раз внимательно главу 3, в частности те разделы, где речь идёт о представлении чисел с плавающей запятой в Python и о проблемах с точностью арифметических действий с этими значениями.

6-й час

Основные типы данных: последовательности и словари

В предыдущей главе Вы узнали о числовых типах данных, с которыми можно работать в Python. Здесь мы продолжим изучение основных типов данных и рассмотрим последовательности (строка символов, набор, список), словари и массивы (вкратце).

Последовательности

Все разновидности этого типа данных отличаются от числовых типов тем, что могут использоваться двумя основными способами. Первый подход аналогичен тем приёмам, которые применяются к числам. Другими словами, можно присваивать переменным значения этих типов, после чего выполнять операции с переменной как с одним целым. Второй подход состоит в том, чтобы рассматривать их как последовательности значений, т. е. области компьютерной памяти изменяющегося размера, которыми можно оперировать как единым целым, а также вычленять из них отдельные элементы, изменять их и снова соединять вместе. В Python имеется возможность создавать последовательности числовых значений. Они называются массивами. Есть даже стандартный модуль, который содержит средства, поддерживающие работу с массивами. Но массивы в Python широко не используются, так как существуют альтернативные более мощные типы последовательностей, для которых разработано больше полезных функций. В конце этой главы я отведу немного времени обсуждению модуля массивов и особенностям его использования.

Строки символов

В Python строки символов (известны как тип *strings*) относятся к основным типам данных (Вы уже видели их в работе в предыдущих главах). Python определяет строки символов как "все, что угодно, заключенное в кавычки". А в кавычки можно заключать всё, что угодно. Большинство компьютерных языков однозначно разграничивает области применения различных видов символов, обозначающих кавычки. Но Python не разменивается на такие мелочи. Одинарные и двойные кавычки обрабатываются одинаково,

поэтому ответ на вопрос, какой вид кавычек лучше использовать, скорее определяется вашими личными симпатиями и привычками.

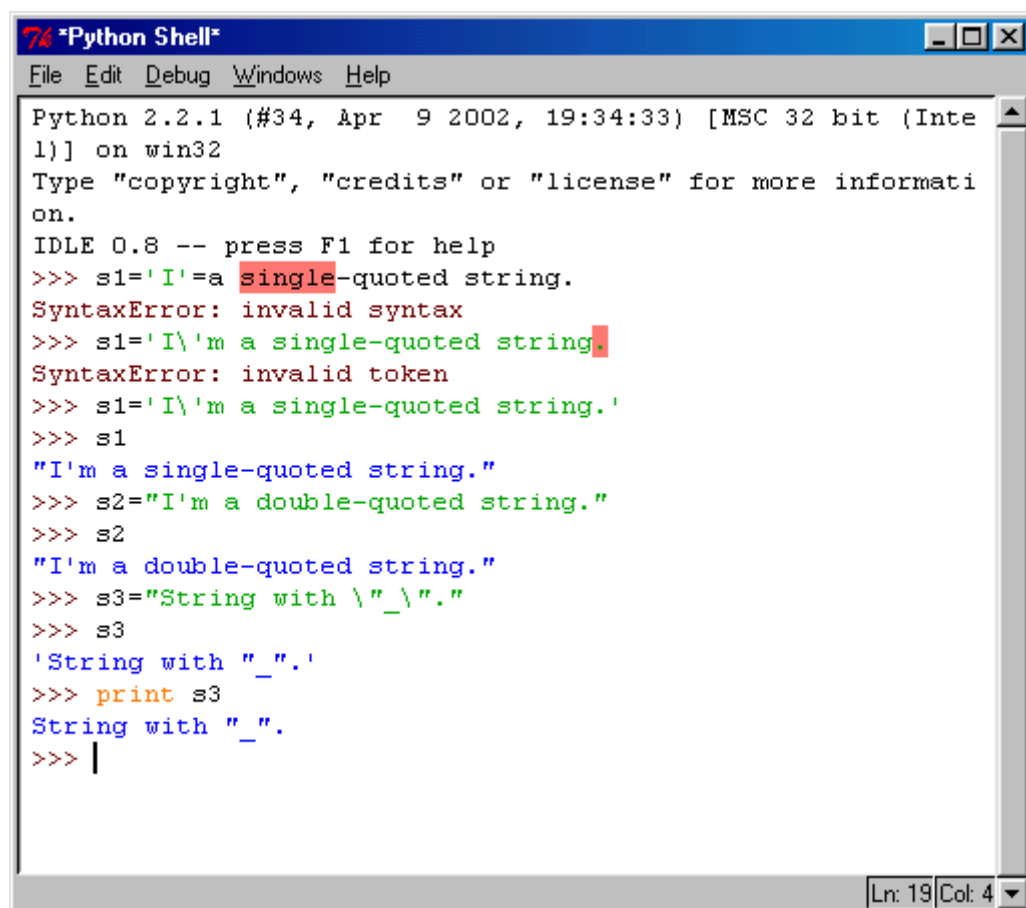
***Прим. В. Шипкова: лично я из ассемблера вынес такое полезное правило – одиночные символы и строки заключать в одинарные кавычки. Чем это удобно? В России (да и зарубежом тоже) принято в тексте использовать двойные кавычки. Так вот до тех пор, пока не встречена интерпретатором закрывающая (парная) одиночная кавычка – все внутренние двойные кавычки будут считаться текстом. ИМХО, это довольно удобно. Кроме того, для выставления одинарной кавычки не нужно нажимать клавишу <Shift>. ;-)**

Какие символы могут стоять между кавычками? Это зависит от конкретной ситуации. Если Вы пользуетесь одинарными кавычками, то между ними можно располагать всё, что угодно, кроме одинарных кавычек. Если Вы предпочитаете двойные кавычки, то в строке нельзя использовать двойные кавычки. Конечно, имеются специальные приёмы, чтобы сообщить Python о том, что те или иные символы не являются служебным, а входят в строку. Символ обратной косой черты (\) является таким управляющим символом, который сообщает интерпретатору, что следующий за ним символ должен быть обработан не так, как обычно. В случае символа кавычки отличие заключается в том, что кавычки больше не являются признаком конца строки, а как её элемент. Немного позже Вы познакомитесь с другими управляющими символами. На рис. 6.1 показано, что произойдет, если, например, попытаться поместить одинарную кавычку внутри строки символов, выделенной такими же одинарными кавычками, а также два метода исправления этой ошибки.

Обратите также внимание на отличия, проявляющиеся в тех случаях, когда для вывода значения на печать используется инструкция `print s1, s2` и когда Вы просто вводите `s2` и нажимаете клавишу <Enter>. При использовании оператора `print` символьная строка, которую он отображает на экране, не заключается в кавычки. Дело в том, что когда Вы вводите имя переменной и нажимаете клавишу <Enter> в диалоговом режиме, Python предполагает, что Вы хотите увидеть, какое значение содержится в переменной, поэтому вставляет двойные кавычки вокруг строковых значений, чтобы напомнить Вам о типе данной переменной. Если же используется оператор `print`, то Python выводит только содержимое переменной, предполагая, что Вы явно укажете, если Вам нужны будут кавычки. Обратите также внимание на то, что текст, набранный национальными символами (в данном случае в

кириллице), вывести в диалоговом режиме можно только командой `print`.

***Прим. В. Шипкова:** опять же это не совсем так. Если не используется кодировка `utf8` (выставляется в настройках IDLE) Вы увидите только кракозябры. Точнее, можно нормально и с другими кодировками работать, но нужно указать с какой именно.



```
Python Shell
File Edit Debug Windows Help

Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> s1='I'=a single-quoted string.
SyntaxError: invalid syntax
>>> s1='I\'m a single-quoted string.'
SyntaxError: invalid token
>>> s1='I\'m a single-quoted string.'
>>> s1
'I'm a single-quoted string.'
>>> s2="I'm a double-quoted string."
>>> s2
'I'm a double-quoted string.'
>>> s3="String with \"_\"."
>>> s3
'String with "_".'
>>> print s3
String with "_".
>>> |
```

Рис. 6.1. Ввод и вывод строк, содержащих символы кавычек и буквы национального алфавита

Кроме использования символов одинарных и двойных кавычек, есть ещё третий способ разграничить строку символов, который реализуется с помощью сочетания, состоящего из трёх идущих подряд двойных или одинарных кавычек. Упоминаемые в повседневной речи как "строки символов, заключённые в тройные кавычки", или просто "тройные кавычки", они используются для того, чтобы помещать в строковые переменные многострочный текст. Чаще всего этот подход используют для создания документации функции или модуля, с чем мы познакомимся подробнее в главе 11. На рис. 6.2 показан ввод строки символов, заключённой в тройные кавычки. На этом примере Вы можете убедиться в преимуществах такого подхода использования кавычек.


```

Python 2.2.1 (#34, Apr  9 2002, 19:34:33) [MSC 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license" for more informatio
n.
IDLE 0.8 -- press F1 for help
>>> s="""Copyright (c) 2001, 2002 Python Software Foundation
.
All rights reserved.

Copyright (c) 2000 BeOpen.com.
All rights reserved."""
>>> print s
Copyright (c) 2001, 2002 Python Software Foundation.
All rights reserved.

Copyright (c) 2000 BeOpen.com.
All rights reserved.
>>>

```

Рис. 6.2. Тройные кавычки

Вывод данной строки будет отличаться в зависимости от того, просто Вы укажете переменную `s`, нажимая клавишу `<Enter>`, или используете инструкцию `print`. Заключение строки символов в тройные кавычки позволяет использовать в ней практически всё символы, включая символы разрыва строки и возврата каретки. Единственное исключение – в строке символов, заключенной в тройные кавычки, не должно быть трёх подряд идущих двойных кавычек. Впрочем, при необходимости можно вставить и их, предварив каждую управляющим символом обратной косой черты, как это было показано на рис. 6.1. Поскольку символ обратной косой черты внутри строки символов всегда имеет особое значение для Python, все эти значения приведены в табл. 6.1.

Таблица 6.1. Управляющие символы, используемые в строках

Символ	Назначение
<code>\</code>	Продолжение строки, если этот символ стоит в конце строки (он должен быть последним символом в строке, поскольку пробел в конце строки после обратной косой черты будет расценен как синтаксическая ошибка)
<code>\\</code>	Обратная косая черта в строке символов

\'	Одинарная кавычка в строке символов
\"	Двойная кавычка в строке символов
\a	Звуковой сигнал (команда компьютеру подать звуковой сигнал)
\b	Команда "забой символа" (аналогична нажатию клавиши <backspace>)
\e	Команда "отменить" (аналогична нажатию клавиши <Esc>; команды \033 или
\x!	В не являются управляющими символами обратной косой черты)
\0	Вставляет значение NOLL (см. текст)
\n	Вставляет символ разрыва строки (<i>linefeed</i>)
\v	Вставляет символ вертикальной табуляции (практически никогда не используется)
\t	Вставляет символ обычной (горизонтальной) табуляции
\r	Вставляет символ возврата каретки (<i>carriage return</i>)
\f	Вставляет символ разрыва страницы (<i>form feed</i>)
\0nn	Вставляет символ восьмеричного значения (см. текст)
\xnn	Вставляет символ шестнадцатеричного значения (см. текст)
\др	Вставляет \др. Другими словами, если значение др не совпадает ни с одним из перечисленных выше символов, Python оставляет в строке последовательность символов \др такой, как она есть. Это аналогично операции тождества в математике

Большинство из указанных выше управляющих символов Вам никогда не понадобится. Так, вертикальная табуляция сейчас практически не применяется, поскольку телетайп, для которого была предназначена эта команда, технически уже полностью устарел и почти нигде не используется. Также Вам вряд ли придётся заботиться о различиях между знаками разрыва строки, возврата каретки и разрыва страницы, если только темой вашей работы не будет перенос текстовых файлов между системами UNIX и DOS или Windows и Macintosh. А если это так, то в таком случае я уверен, Вы уже знаете более чем достаточно об этих различиях. В листинге 6.1 представлен код небольшой программы, которая демонстрирует применение некоторых наиболее распространённых управляющих символов.

***Прим. В. Шипкова: код подачи звукового сигнала вряд ли будет работать. По крайней мере, у меня точно не работает.**

Листинг 6.1. Управляющие символы (программа *xtst.py*)

```
#!c:\python\python.exe
```

```

s="e:\\Beginner"
s1="e:" "\\ " "Beginner"
s2=s1+\\
"\\tst.py"

print "This is a DOS path:", s
print "This is a DOS path:", s1
print "This is a DOS path:", s2

s3="I contain','single' quotes"
s4="I contain \"double\" quotes"
s5="""I am a triple-quoted string that contains "\\\"\\\"
quotes"""

print s3
print s4
print s5

s6="I contain\\t\\t\\tthree\\t\\t\\ttabs"
s7="I contain a\\t\\v\\tvertical tab"
s8="I contain a\\t\\a\\tBELL, which you can hear"

print s6
print s7
print s8

s9="I contain a BACK\\bSPACE"
s10="I contain a BACKK\\bSPACE AND a \\nNEWLINE and a
\\rLINEFED"
s11="I've got a FORM\\fFEED!"

print s9
print s10
print
print s11

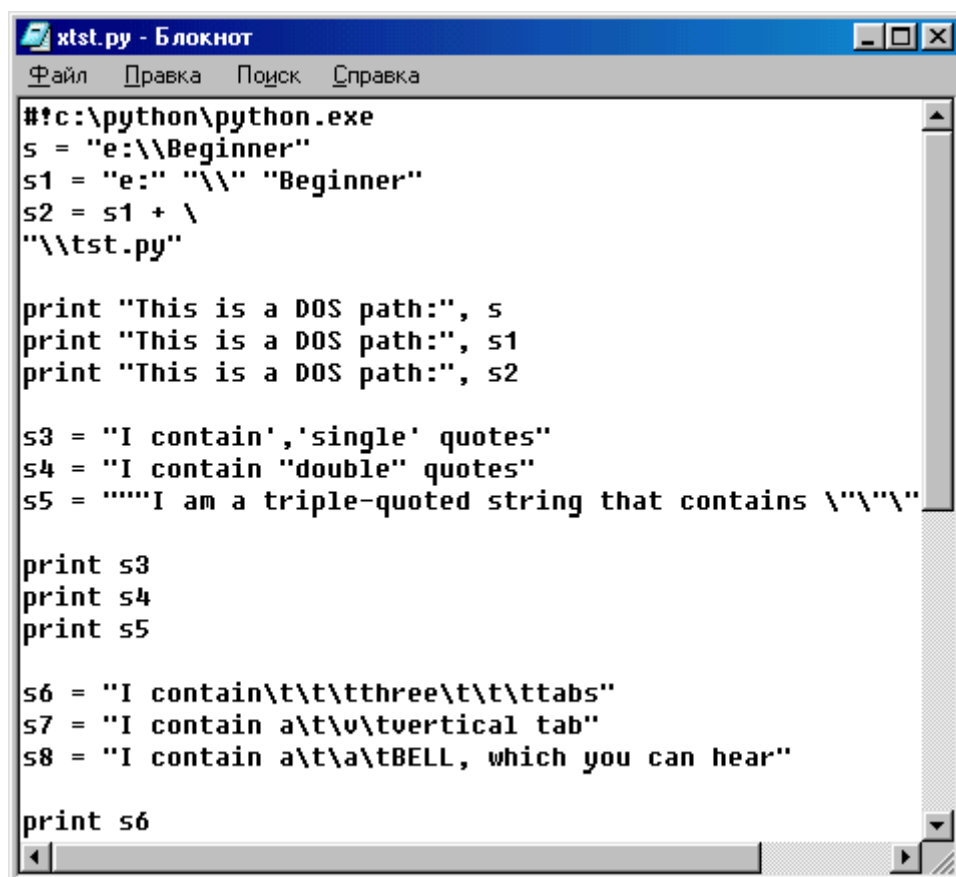
s12="If Python doesn't know what the escape code means,\\n"
\\
"it performs the identity operation! \\identity!"
s13="But if you don't know what a code means, don't use
it!"

print s12
print s13

```

На рис. 6.3 показан результат выполнения программы `xtst.py` в окне DOS, а на рис. 6.4 – результат выполнения той же программы, но уже в окне IDLE. Как видите, во втором

случае управляющие символы отображаются не так, как в окне DOS.



```
#!c:\python\python.exe
s = "e:\\Beginner"
s1 = "e:" "\\ " "Beginner"
s2 = s1 + \
"\\tst.py"

print "This is a DOS path:", s
print "This is a DOS path:", s1
print "This is a DOS path:", s2

s3 = "I contain','single' quotes"
s4 = "I contain \"double\" quotes"
s5 = """"I am a triple-quoted string that contains \\\"\\\"\\\"

print s3
print s4
print s5

s6 = "I contain\\t\\t\\tthree\\t\\t\\ttabs"
s7 = "I contain a\\t\\v\\tvertical tab"
s8 = "I contain a\\t\\a\\tBELL, which you can hear"

print s6
```

Рис. 6.3. Результат выполнения программы xtst.py в окне Блокнота

```
7% *xtst.py - C:\WINDOWS\???????? ????*xtst.py*
File Edit Windows Help

#!C:\python\python.exe
s = "e:\\Beginner"
s1 = "e:" "\\ " "Beginner"
s2 = s1 + \
"\\tst.py"

print "This is a DOS path:", s
print "This is a DOS path:", s1
print "This is a DOS path:", s2

s3 = "I contain','single' quotes"
s4 = "I contain "double" quotes"
s5 = """I am a triple-quoted string that contains \"\" \" quote

print s3
print s4
print s5

s6 = "I contain\t\t\tthree\t\t\ttabs"
s7 = "I contain a\t\v\tvertical tab"
s8 = "I contain a\t\a\tBELL, which you can hear"

print s6
print s7
print s8

s9 = "I contain a BACK\bSPACE"
s10 = "I contain a BACK\bSPACE AND a \nNEWLINE and a \rLINEFE"
s11 = "I've got a FORM\fFEED!"

Ln: 1 Col: 3
```

```
7% 1.py - C:\??? ??????????1.py
File Edit Windows Help

#! c:\python\python.exe
s = "e:\\Beginner"
s1 = "e:" "\\ " "Beginner"
s2 = s1 + \
"\\tst.py"
print "This is a DOS path:", s print "This is a DOS path:", s1 }
s3 = "I contain 'single' quotes" s4 = 'I contain "double" quote:
print s3 print s4 print s5
s6 = "I contain\t\t\tthree\t\t\ttabs"
s7 = "I contain a\t\v\tvertical tab"
s8 = "I contain a\t\a\tBELL, which you can hear"
print s6 print s7 print s8
s9 = "I contain a BACK\bSPACE"
s10 = "I contain a BACK\bSPACE AND a \nNEWLINE and a XrLINEFEE"
s11 = "I've got a FORM\fFEED!"
print s9 print s10 print print s11
s12 = "If Python doesn't know what the escape code means,\n" \
"it performs the identity operation! \identity!"
s13 = "But if you don't know what a code means, don't use it!"
print s12 print s13

Ln: 1 Col: 0
```



Рис. 6.4. Результат выполнения программы `xtst.py` в окне `IDLE`

Причина несовпадения символов объясняется тем, что в DOS используется расширенный набор символов ASCII. В листинге 6.2 показана ещё одна небольшая программа на языке Python (`ascii.py`), которая выводит таблицу символов ASCII Вашего компьютера. Ее можно выполнять на любой платформе, которая использует стандарт ASCII.

Термин ASCII является аббревиатурой от слов *American Standard Code for Information Interchange* (Американский стандартный код для обмена информацией). Первоначально этот стандарт предусматривал только 7-битовый код, но затем он был расширен до 8 битов. Таблица ASCII для 7-битовых кодов содержит 128 символов от 0000000 до 1111111, а таблица для 8 битовых кодов простирается до 11111111. В шестнадцатеричном виде это составляет ряд от 00 до FF.

```
#!c:\python\python.exe

i = 0
while i < 256 :
    print chr(i)
    if i != 0 and i % 8 == 0:
        print
    i = i + 1
```

Функция `chr()` встроена в Python. Она принимает число в аргументе и возвращает соответствующий символ из таблицы ASCII. Обратная функция, `ord()`, в качестве аргумента принимает символ, а возвращает его номер в таблице.

На рис. 6.5 показано выполнение программы `ascii.py` в окне DOS.

```
Python Shell
File Edit Debug Windows Help

>>> i=0
>>> while i<256:
    print chr(i),
    if i!=0 and i%8==0:
        print
    i=i+1

  □ □ □ □ □ □ □ □
□ □ □ □ □ □ □ □
□ □ □ □ □ □ □ □
! " # $ % & ' (
) * + , - . / 0
1 2 3 4 5 6 7 8
9 : ; < = > ? @
A B C D E F G H
I J K L M N O P
Q R S T U V W X
Y Z [ \ ] ^ _ `
a b c d e f g h
i j k l m n o p
q r s t u v w x
y z { | } ~ □ □
□ □ □ □ □ □ □ □
□ □ □ □ □ □ □ □
□ □ □ □ □ □ □ □
□ □ □ □ □ □ □ □
i o £ ¤ ¥ ¦ § ¨
© º « ¬ ® ¯ °
± ² ³ ´ µ ¶ · ¸
¹ º » ¼ ½ ¾ ¿ À
Á Â Ã Ä Å Æ Ç È
É Ê Ë Ì Í Î Ï Ð
Ñ Ò Ó Ô Õ Ö × Ø
Ù Ú Û Ü Ý Þ à
á â ã ä å æ ç è
é ê ë ì í î ï ð
ñ ò ó ô õ ö ÷ ø
ù ú û ü ý þ ÿ
>>> |
```

Рис. 6.5. Вывод таблицы символов ASCII с помощью программы `ascii.py` в окне DOS

***Прим. В. Шипкова: конечно же на рисунке не окно DOS, а самый что ни на есть IDLE.**

Иногда управляющие символы в тексте могут оказаться не столько полезными, сколько надоедающими. Например, в системе Windows часто бывает удобно поместить в переменную описание пути к файлу. Однако в Windows описания пути содержат символы обратной косой черты, которые служат

разделителями каталогов в DOS и Windows. Необходимость вводить каждый раз этот символ дважды утомляет. Если же забыть об этом и ввести только одну обратную косую черту, то результат выполнения программы будет непредсказуем, а ошибку будет трудно найти. Для решения этой проблемы Guido ввёл в Python средство установкинеобрабатываемых строк. Чтобы сообщить Python, что данная строка символов является необрабатываемой, необходимо поместить перед ней букву "r" (r происходит от английского названия *row string*). Например, вот так: `s = r"k:\Beginner"`

Необрабатываемые строки используются и во многих других случаях, но главная причина, почему Guido ввёл их в Python, состоит в том, что они намного упрощают написание регулярных выражений. Без них создание регулярных выражений вылилось бы просто в титанический труд. Регулярные выражения не рассматриваются в этой книге, но Вы можете найти подробную информацию о них в Internet по адресу <http://www.Python.org/>.

Строки символов удобно использовать в программах, принимающих данные, вводимые пользователем с клавиатуры, для документирования работы функции или модуля, для создания дружественного пользовательского интерфейса, а также во время отладки программы. Возможно, Вы уже знаете, что те же операторы, которые мы использовали с числовыми значениями, например оператор суммирования (+), могут также использоваться для работы со строками, но при этом они выполняют другие действия. И это не случайно. Python зорко следит за типом переменных, так что в любой момент он точно знает, какой именно реакции Вы от него ожидаете. Например, `1+1` совершенно очевидно представляет математическую инструкцию. А что подразумевается под "`строка1`" + "`строка2`"? Безусловно, знак + в данном случае означает что-то другое. В контексте строк символ + означает объединение двух последовательностей символов в одну (по-научному эта операция называется *конкатенацией*). Результатом действия "`строка1`" + "`строка2`" будет строка "`строка1строка2`". В зависимости от контекста такая возможность использовать один и тот же оператор для выполнения различных действий называется *перегрузкой оператора*. При умелом применении эта возможность может оказаться одним из наиболее ценных средств программирования Python. Немного позже, по мере усвоения материала, Вы научитесь выполнять перегрузку собственных операторов.

В табл. 6.2 приведён список операторов, которые можно использовать для выполнения операций над строками. Далее мы

рассмотрим подробнее все эти операторы в порядке их следования в таблице.

Таблица 6.2. Операторы, используемые для обработки строк СИМВОЛОВ

Оператор	Значение	Пример
<code>s1+s2</code>	Конкатенация строк <code>s1</code> и <code>s2</code>	<code>s3 = s1 + s2</code>
<code>s*n</code> или <code>n*s</code>	Множественное повторение строки <code>s</code> – повторить строку <code>s</code> <code>n</code> раз	<code>t = s * 5</code>
<code>u in s</code> <code>u not in s</code>	Является ли <code>u</code> элементом последовательности <code>s</code> или не является?	<code>if u in s:</code> инструкции <code>for u in s:</code> инструкции
<code>s[n]</code>	Возвращение элемента последовательности <code>s</code> по индексу <code>n</code>	<code>z = s[0]</code>
<code>s[i:j]</code>	Извлечение из строки <code>s</code> последовательности символов от <code>i</code> до <code>j</code>	<code>z = s[0:2]</code>
<code>len(s)</code>	Возвращение длины строки <code>s</code>	<code>l = len(s)</code>
<code>min(s)</code>	Возвращение минимального элемента последовательности <code>s</code>	<code>m = min(s)</code>
<code>max(s)</code>	Возвращение максимального элемента последовательности <code>s</code>	<code>m = max(s)</code>

Мы уже видели пример использования оператора `+`. Использование Оператора повторения `*` также вполне очевидно (рис. 6.6).

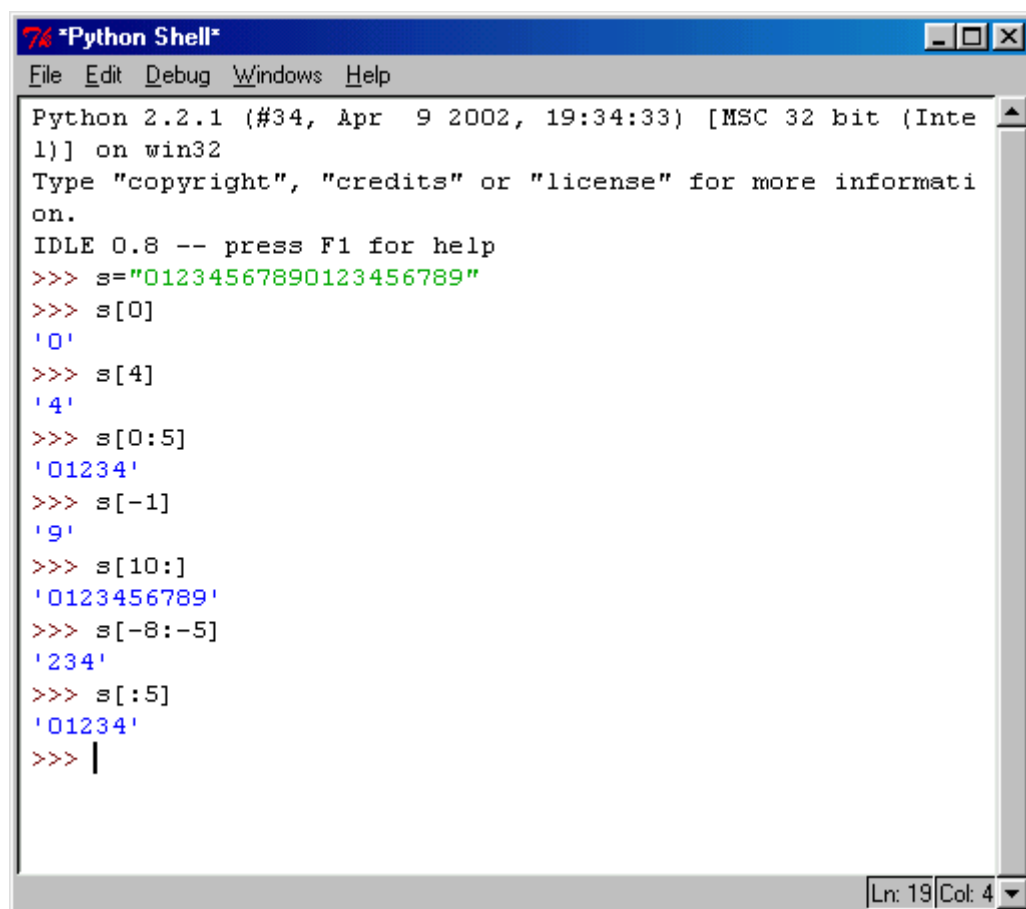
```

Python 2.2.1 (#34, Apr  9 2002, 19:34:33) [MSC 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license" for more informatio
n.
IDLE 0.8 -- press F1 for help
>>> a="a:"
>>> t=8*a
>>> print a, t
a: a:a:a:a:a:a:a:a:
>>> |
  
```

Рис. 6.6. Оператор повторения

Оператор индексирования `[]` позволяет извлекать из строк отдельные символы или последовательности символов. Работа этого оператора не зависит от того, какие символы

используются в строке. Так, на рис. 6.7 показано извлечение символов из строки цифр.



```
Python 2.2.1 (#34, Apr  9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> s="01234567890123456789"
>>> s[0]
'0'
>>> s[4]
'4'
>>> s[0:5]
'01234'
>>> s[-1]
'9'
>>> s[10:]
'0123456789'
>>> s[-8:-5]
'234'
>>> s[:5]
'01234'
>>> |
```

Рис. 6.7. Оператор индексирования

В отличие от многих других компьютерных языков, Python позволяет использовать отрицательную индексацию, т.е. ввод отрицательных индексов в принципе допускается, но результат будет непредсказуемым. Яркий тому пример — язык C. Для Python отрицательная индексация означает отсчёт символов в направлении от конца последовательности к началу. Ту же операцию можно выполнить более изящно: найти сначала размер последовательности, а затем вычесть из полученного значения требуемое число. Например, если нужно извлечь последний элемент строки символов, можно использовать следующую инструкцию:

```
e = s[len(s) - 1]
```

Если Вы забудете отнять единицу, то Python сообщит, что индекс вышел за границы диапазона. Автоматическое отслеживание в Python границ последовательностей — это ещё одна полезная особенность данного языка программирования. Другие языки такой информацией не владеют и безразлично реагируют на Ваши попытки возвратить элементы последовательности, которых просто не существует. Как



правило, такие попытки приводят к серьезным ошибкам в ходе выполнения программы, которые довольно сложно обнаружить и исправить в коде программы. Подобные ошибки случаются настолько часто, что Guido посчитал своей обязанностью сделать что-то для их предупреждения.

Использование индексов, выходящих за пределы последовательности, не только не приводит в Python к ошибкам, но может оказаться даже полезным, например, когда нужно извлечь или скопировать всю строку. Для этого не нужно много размышлять над тем, какую длину имеет та или иная строка. Например, если есть строка `s` длиной 10 символов, то все следующие инструкции `s[:]`, `s[:2000]`, `s[-2000:2000]` и `s[-2000:]` выполняют одно и то же действие – копируют всю последовательность целиком. (Число 2000 в данном случае использовалось для примера. Вы можете вставить любое другое большое значение.) Фактически оператор `s[:]` в исходном коде Python реализован с применением очень больших значений индексов начала и конца последовательности. Только в тех случаях, когда нужно извлечь конкретный элемент последовательности, указание индексов должно быть точным.

В других языках также имеются функции, позволяющие извлекать подстроки (части строк символов), но ни один из них не предоставляет столько удобств, как Python. (Что-то подобное умеет делать и Perl, но лично мне его синтаксис кажется не столь очевидным.) Примите к сведению, что для индексирования элементов и извлечения подстрок можно использовать не только любые целые числа, но и любые целочисленные переменные, что позволяет программно управлять размером извлекаемых подстрок.

Если оператор индексирования открывает доступ к элементам последовательности, то нельзя ли с его помощью выполнять присвоение значений элементам последовательности? Ответ однозначный – нет. Символьные строки являются неизменяемыми, Вы не можете изменять их после того, как создали. Это означает, что категорически запрещаются действия, подобные следующему:

```
s = "Where is misstake in this string."  
s[-10] = 'н'
```

Python будет бурно возмущаться, если Вы попытаетесь подсунуть ему подобный код.

***Прим. В. Шипкова: это также справедливо и для версии 2.4.1**

Способ, позволяющий изменить строку символов, состоит не в том, чтобы изменить её (вот Вам и немного дзен-философии), а в том, чтобы создать новую. В приведённом выше примере ошибку можно исправить следующим образом:

```
s="Where is misstake in this string."  
s1=s[:-10]  
s2=s[-9:]  
s=s1+'н'+s2
```

Выполнение этого кода в окне IDLE показано на рис. 6.8.

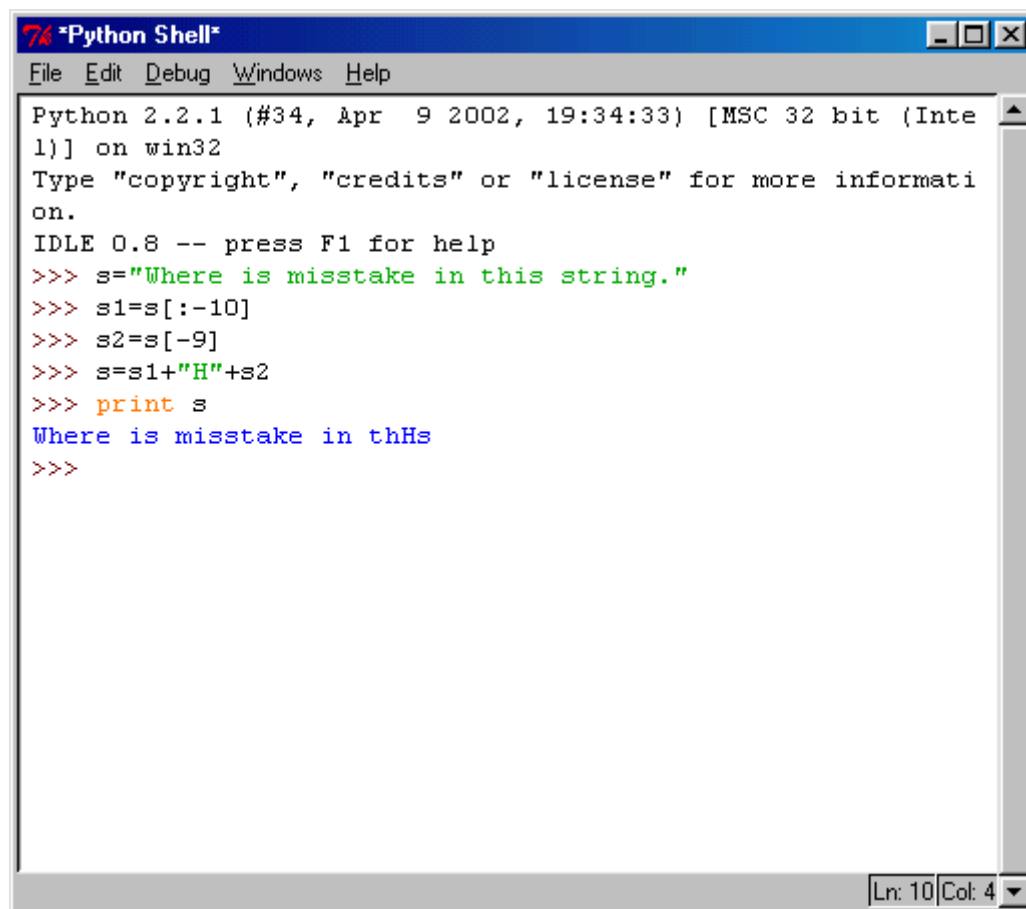


Рис. 6.8. Изменение строки

Последние средства обработки последовательностей - функции `min()` и `max()`. Эти функции извлекают из любой последовательности, переданной им в аргументе, соответственно самый маленький или самый большой элементы. При работе со строками символов минимальный и максимальный элементы однозначно определяются алфавитным порядком следования символов. Другими словами, буква А "меньше" буквы В. Другие операторы сравнения, с которыми мы сталкивались раньше, работают по тому же принципу. Если быть более точным, то определяющим является порядок следования символов в копировочной таблице компьютера, где для символов букв соблюдается алфавитный порядок. Но с

помощью данных функций и операторов сравнения можно сравнивать также символы, не относящиеся к буквам алфавита, либо символы, относящиеся к разным алфавитам. Так, латинские буквы всегда "меньше" букв кириллицы. В отличие от других языков в Python могут непосредственно сравниваться ещё и строки целиком. Поэтому вполне допустимы выражения следующего вида:

```
if "Сергей" > "Олег" :  
    инструкции
```

Сравнение переменных, содержащих строки символов, правомерно, как и сравнение переменных, содержащих числа. При этом всегда следует помнить, что прописные буквы в таблице ASCII всегда "меньше" строчных, иначе Вы можете получить результаты, которые сильно поразят Вас. Один из методов обойти подобные неожиданности состоит в том, чтобы всегда сравнивать строчные буквы, используя метод `string.lower()` для приведения всех алфавитных символов к нижнему регистру.

Пришло время ещё раз рассмотреть оператор `in`, с которым мы уже познакомились в главе 4. В Python цикл `for` не подчиняется тем же самым правилам, которые установлены в других языках программирования. Его синтаксис имеет следующий вид:

```
for <целевая_переменная> in <список> :  
    инструкции  
else:  
    инструкции "выполняются только в том случае,  
    если цикл не был прерван оператором break"
```

Так вот, в качестве компонента `<список>` может использоваться любая последовательность. Поскольку строки символов являются последовательностями, то мы вправе использовать их в операторах `for`. Предположим, что в программе допускается ввод пользователем пути к файлу и нам необходимо удостовериться, что эта строка представляет собой допустимое описание пути в системе DOS. На рис. 6.9 демонстрируется простой способ выполнения такой проверки.

```

Python 2.2.1 (#34, Apr  9 2002, 19:34:33) [MSC 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license" for more informatio
n.
IDLE 0.8 -- press F1 for help
>>> s=r"c:\Python\tests\spam.py"
>>> for i in s:
        if i==";":
            print "ok"
            break

ok
>>> if ':' in s:
        print "ok"

ok
>>> if '\\' in s:
        print "ok"

ok
>>>

```

Рис. 6.9. Проверка правильности пути DOS

Если Вы внимательно проанализируете задачу, то придёте к выводу, что особой необходимости в применении цикла for тут нет. Простого теста на принадлежность-непринадлежность было бы достаточно. Тем не менее применение цикла for довольно часто оказывается полезным, а в данном случае мы просто рассмотрели пример его использования.

Вам могут понадобиться ещё и другие средства преобразования строк, например преобразование символа в число, но для выполнения подобных действий используются не операторы, а специальные функции, определённые в модуле string. Чтобы использовать данный модуль, его следует импортировать где-то в начале (лучше в самом начале) кода программы. В табл. 6.3 приведены наиболее важные и широко используемые функции обработки строк из модуля string.

Таблица 6.3. Основные функции обработки строк

Функция	Описание
atoi(s[,base])	Преобразовывает s в целое число, используя необязательный параметр base для установки основания системы счисления (по умолчанию — 10). Возвращает целое число

<code>atoi(s[,base])</code>	Преобразовывает <i>s</i> в длинное целое число, используя необязательный параметр <i>base</i> как основание системы счисления. Возвращает длинное целое число
<code>atof(s)</code>	Преобразовывает <i>s</i> в число с плавающей запятой (<i>s</i> может содержать показатель степени). Возвращает число с плавающей запятой
<code>split(s[,sep[,maxsplit]])</code>	Просматривает <i>s</i> в поисках символов, указанных в <i>sep</i> , и разбивает <i>s</i> на слова. Возвращает список слов. Если аргумент <i>sep</i> не указан, его значением по умолчанию является пробел. Если присутствует параметр <i>maxsplit</i> , то количество разбиений на слова ограничено этим числом, причём последнее слово в возвращаемом списке будет содержать оставшуюся неразбитую часть <i>s</i>

<code>join(words[,sep])</code>	Объединяет все слова списка в одну строку символов, при этом слова отделяются друг от друга символом, указанным в <i>sep</i> . Полученная таким образом строка и возвращается. Если параметр <i>sep</i> отсутствует, по умолчанию символом разбиения устанавливается пробел
<code>find(s,sub[,start[,end]])</code>	Просматривает <i>s</i> в поисках символа, указанного в <i>sub</i> . Если находит, возвращает индекс первого символа <i>sub</i> , обнаруженного в <i>s</i> . Параметры <i>start</i> и <i>end</i> необязательны, но если присутствуют, то <i>start</i> является индексом, указывающим в строке <i>s</i> точку начала поиска, а <i>end</i> является индексом, указывающим в строке точку окончания поиска

Программа `Ibm.py`, приведенная в листинге 6.3, демонстрирует применение некоторых из этих функций. Эту программу я написал, чтобы упорядочить свою коллекцию видеофильмов. Я пронумеровал свои ленты в шестнадцатеричном

формате, начиная с кассеты под номером 0000 (мой шурин считает, что это проявление окончательного помешательства под влиянием компьютеров, но, видимо, я уже не могу мыслить иначе). У меня есть небольшой принтер для печати этикеток, поэтому я могу позволить себе удовольствие напечатать самоклеющиеся этикетки для футляров кассет. Я могу распечатать целую стопку за один раз, обеспечив при этом соблюдение определённого формата в соответствии с шаблоном, сохранённым в текстовом файле. Приведенная ниже программа выводит на печать числа в нужном формате, либо по одному за один раз, либо начиная с номера, указанного в виде параметра в командной строке. Сначала просмотрите эту программу сами, а затем мы подробно её обсудим.

Листинг 6.3. Программа Ibm.py

```
#!c:\python\python.exe
import sys
import string

def printlabel(n):
    s="%04X"%(n)
    sys.stdout.write("%s\r\n"%(s))
    for i in s:
        sys.stdout.write("%s\r\n"%(i))
        sys.stdout.write("\r\n")
        sys.stdout.flush()

if __name__ == "__main__":
    if len(sys.argv)>1:
        if sys.argv[1]=="-h":
            print """Usage: Ibm start_number quantity
Example:
Ibm B30 100
Start_number is hex, quantity is decimal
if no arguments are given, Ibm reads from
stdin."""
            sys.exit(0)
        elif len(sys.argv)>2:
            n = string.atoi( sys.argv[ 1 ], 0x10 )
            e = string.atoi( sys.argv[ 2 ] )
            for n in range( n, n+e ):
                printlabel(n)
        else:
            while 1:
                p=sys.stdin.readline( )
                if not p:
                    break
                n=string.atoi(p, 0x10)
```

`printlabel (n)`

***Прим. В. Шипкова: вразумительного ответа от этого примера Вы скорее всего не добьётесь, так как печать сообщений происходит в стандартный поток. Но тем не менее заменив все выходы `sys.stdout` на `print` можно посмотреть, что будет. ;-)**

Функция `printlabel()` принимает целочисленный аргумент `n`, который сначала преобразовывается в строку шестнадцатеричных символов (инструкции, содержащие `%04X`, мы рассмотрим подробно в главе 8), после чего особым образом выводится результат (`sys.stdout.write()`). Затем запускается цикл `for i in s` (здесь `s` является последовательностью), который записывает каждую шестнадцатеричную цифру в своей собственной строке, а в конце вставляет пустую строку. В главной части программы (часть кода, которая следует после инструкции `if __name__...`) выполняется проверка ввода пользователем каких-либо параметров при запуске программы. Если параметры присутствуют, то программа считывает первый параметр как шестнадцатеричное значение (инструкция `n = string.atoi (sys.argv [1], 0x10)`) и второй параметр — как десятичное число (инструкция `e=string.atoi (sys.argv [2])`). Только после этого для каждого числа, принадлежащего диапазону, указанному пользователем (инструкция `for n in range (n, n + e):`), вызывается функция `printlabel()`. Надеюсь, Вы не забыли, что `0x10` — это шестнадцатеричное число, а `16` — десятичное.

Если при запуске программы параметры отсутствуют, то `Ibm.py` предоставляет возможность пользователю вводить числа в шестнадцатеричном виде. Каждый номер, который получается таким образом, программа выводит на печать с помощью функции `printlabel()`. Наилучший способ понять, как работает эта программа, состоит в том, чтобы проверить её в работе (как и в отношении любой программы, приведенной здесь, Вы можете загрузить её с Web-страницы данной книги). Введите в командной строке каталога, в котором Вы сохранили данную программу (безусловно, под именем `Ibm.py`), следующую команду:

```
python Ibм.py > out.txt
```

Затем введите одно или несколько шестнадцатеричных чисел и нажмите комбинацию клавиш `<Ctrl+Z>`, которая прервет работу Python (`<Ctrl+D>` — для UNIX). Откройте и просмотрите полученный файл `out.txt`.

Чтобы больше узнать о модуле обработки строк и его функциях, обратитесь к документации, на Python, которую можно найти и загрузить по адресу <http://www.Python.org>.

Наборы

Теперь, после того как мы столь тщательно рассмотрели строки символов, со следующим типом данных у Вас не должно возникнуть никаких проблем. *Наборы (tuples)* представляют собой другую разновидность неизменяемых последовательностей, но они более масштабные по сравнению с обычными строками символов. В строках последовательность состоит из отдельных символов, тогда как в наборах в качестве отдельных элементов могут выступать строки символов, числа любого типа, другие наборы, в общем, любая разновидность данных любого допустимого типа. Сюда входят и определяемые пользователем типы данных, которые будут рассмотрены в части III. В действительности Вы уже сталкивались с наборами ранее в этой книге. Данный раздел поможет закрепить и систематизировать ваши знания.

***Прим. В. Шипкова: также наборы в различных источниках могут упоминаться как кортежи.**

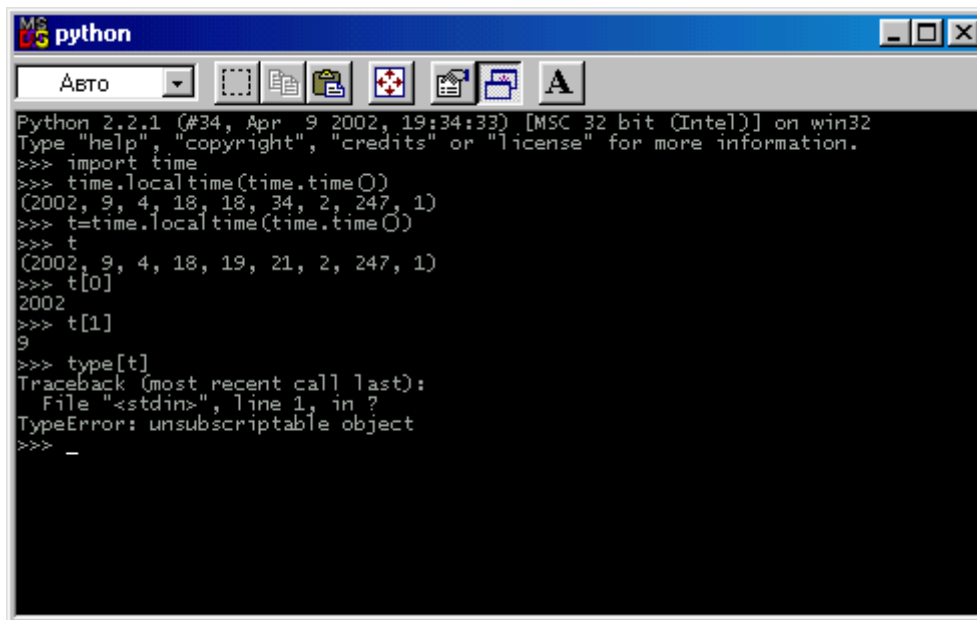
Наборы заключаются в круглые скобки и так же, как и строки символов, неизменяемы. Следующий пример показывает, как сообщить Python, что некоторая переменная является набором:

```
t="Это", "набор", "из", 5, "элементов."  
e=() # Это пустой набор
```

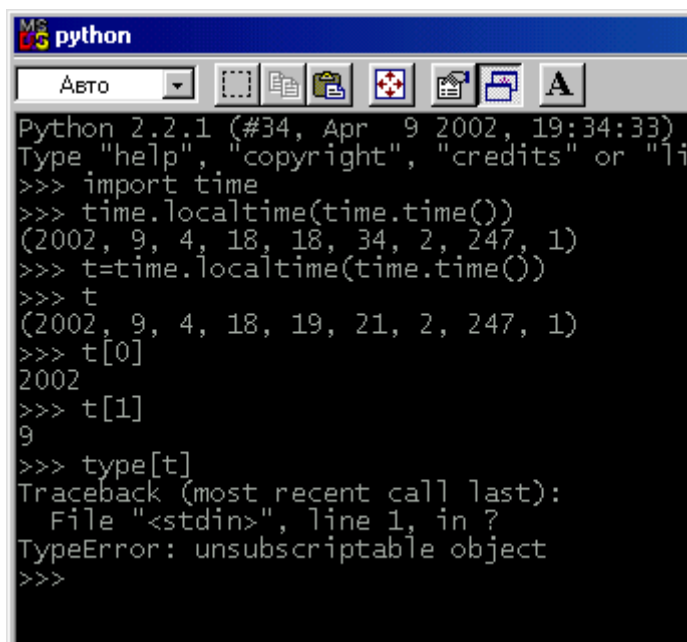
Подобно другим последовательностям, в кортежах можно использовать индексацию: `u=t[-1]`

***Прим. В. Шипкова: вот пожалуйста. Вместо слова набор используется слово кортеж.**

Однако, в отличие от строк, для работы с кортежами отсутствуют какие-либо специальные функции. Нет никакого модуля `tuple`, который можно было бы импортировать. Тем не менее многие функции возвращают результат в виде набора. Прекрасные примеры тому – функции времени и даты. На рис. 6.10 показано несколько примеров использования функции времени и приёмы работы с наборами.



```
python
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import time
>>> time.localtime(time.time())
(2002, 9, 4, 18, 18, 34, 2, 247, 1)
>>> t=time.localtime(time.time())
>>> t
(2002, 9, 4, 18, 19, 21, 2, 247, 1)
>>> t[0]
2002
>>> t[1]
9
>>> type[t]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsubscriptable object
>>> _
```



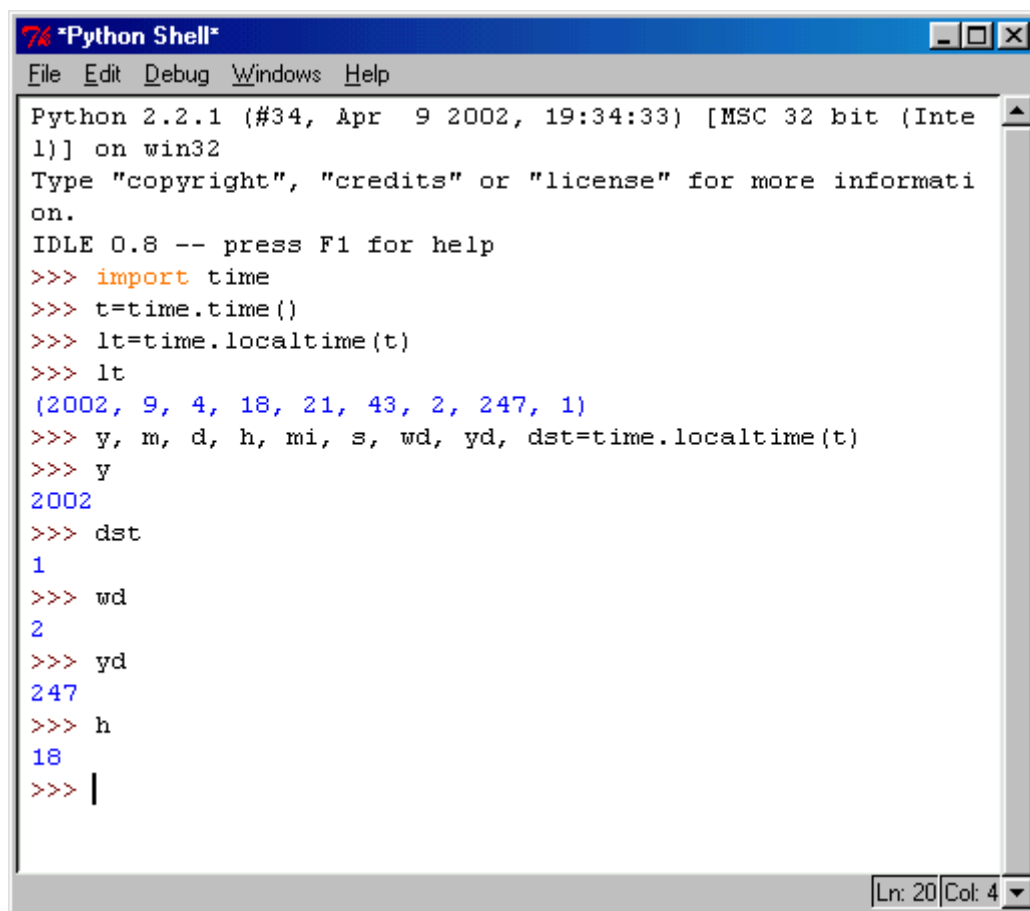
```
python
Python 2.2.1 (#34, Apr 9 2002, 19:34:33)
Type "help", "copyright", "credits" or "li
>>> import time
>>> time.localtime(time.time())
(2002, 9, 4, 18, 18, 34, 2, 247, 1)
>>> t=time.localtime(time.time())
>>> t
(2002, 9, 4, 18, 19, 21, 2, 247, 1)
>>> t[0]
2002
>>> t[1]
9
>>> type[t]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsubscriptable object
>>>
```

Рис. 6.10. Функция времени возвращает набор значений

Python обладает прекрасным качеством, которым могут похвастать немногие языки программирования. заключается оно в умении "распаковывать" наборы, т.е. извлекать содержащиеся в нем значения и присваивать их переменным. В примере, показанном на рис. 6.10, мы присвоили переменной `t` результат выполнения функции `localtime()`, а затем проверили тип этой переменной. Python показал, что это переменная типа `tuple` (набор). Примеры простых, но очень полезных операций, которые можно выполнять с наборами, показаны на рис. 6.11.

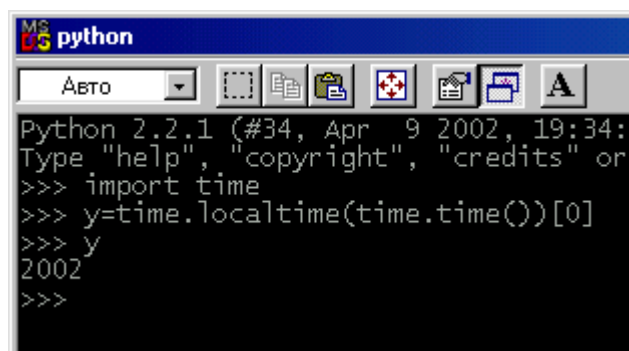
Единственное необходимое условие состоит в том, что количество отделённых запятыми переменных, стоящих слева от оператора присваивания (`=`), должно соответствовать числу

элементов в наборе, возвращённом функцией `localtime()`. Если нужно извлечь только значение года, можно поступить так, как показано на рис. 6.12.



```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> import time
>>> t=time.time()
>>> lt=time.localtime(t)
>>> lt
(2002, 9, 4, 18, 21, 43, 2, 247, 1)
>>> y, m, d, h, mi, s, wd, yd, dst=time.localtime(t)
>>> y
2002
>>> dst
1
>>> wd
2
>>> yd
247
>>> h
18
>>> |
```

Рис. 6. 11. Извлечение значений из набора



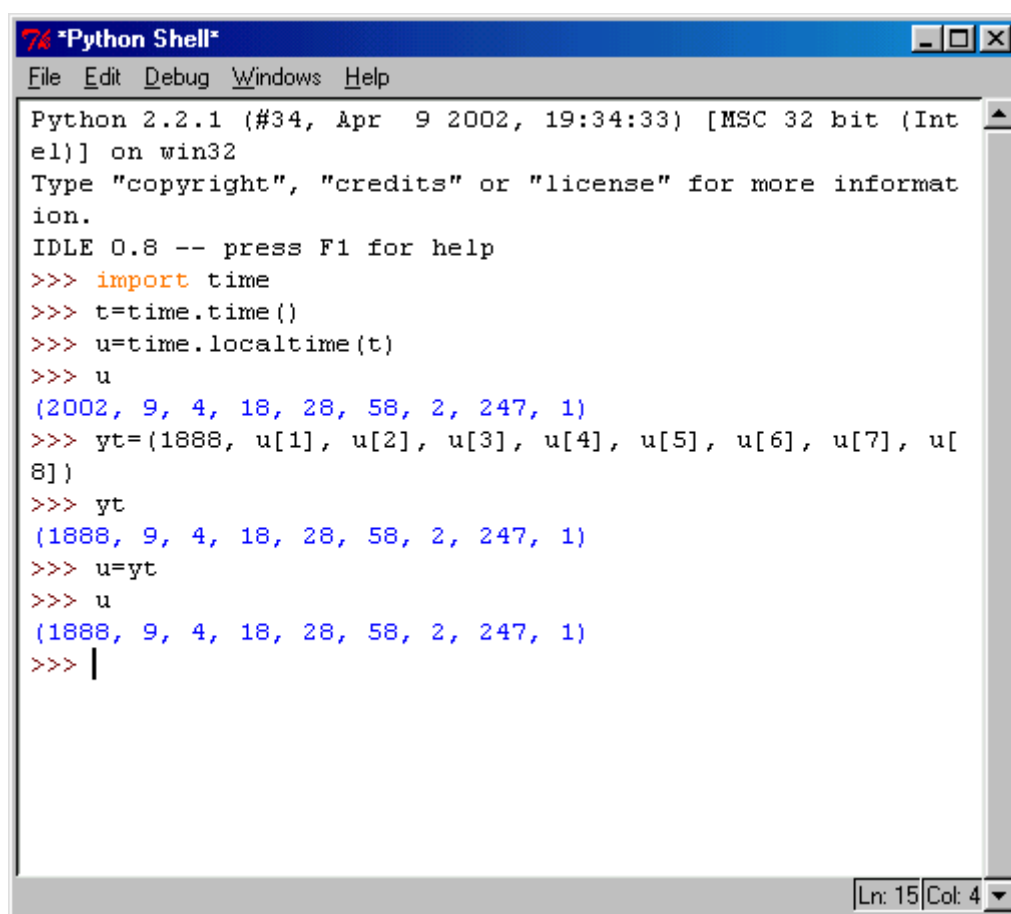
```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import time
>>> y=time.localtime(time.time())[0]
>>> y
2002
>>>
```

Рис. 6.12. Извлечение значения года

В данном случае распаковка всего набора не производится, а извлечение требуемого значения выполняется с помощью оператора индексирования. Функция `time.time()` возвращает время в виде количества секунд, отсчитанных с 1 января 1970 года. Это значение выражено в формате числа с плавающей запятой. Функция `time.localtime()` принимает это число с плавающей запятой в качестве аргумента и возвращает набор из 9 элементов, что мы и наблюдали на рис. 6.11. Во втором примере мы извлекаем из набора элемент с индексом `[0]` (год)

и присваиваем значение переменной `u`. Этот пример демонстрирует, что любой набор (как и любая другая последовательность) может быть индексирован. А оператор индексирования можно использовать не только с самими последовательностями, но и с функциями, которые их возвращают.

Как и в случае со строками, единственный способ, с помощью которого можно изменить набор, состоит в создании нового набора. Например, если необходимо изменить год в наборе значений времени, следует поступить так, как показано на рис. 6.13.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following code and output:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> import time
>>> t=time.time()
>>> u=time.localtime(t)
>>> u
(2002, 9, 4, 18, 28, 58, 2, 247, 1)
>>> yt=(1888, u[1], u[2], u[3], u[4], u[5], u[6], u[7], u[8])
>>> yt
(1888, 9, 4, 18, 28, 58, 2, 247, 1)
>>> u=yt
>>> u
(1888, 9, 4, 18, 28, 58, 2, 247, 1)
>>> |
```

The status bar at the bottom right shows "Ln: 15 Col: 4".

Рис. 6.13. Изменение значения в наборе

Но есть ещё более простой способ, который состоит в том, чтобы использовать конкатенацию строки символов в сочетании с извлечением части строки: `yt = (1888,) + u[1:8]`

Так как в Python очень просто перепутать число, заключенное в круглые скобки, `(1888)`, с набором, содержащим единственный элемент, то добавление запятой после значения `(1888,)` является единственным способом указать Python, что речь идёт именно о наборе. Запятая в данном случае служит условным ключом, сигнализирующим, что это набор. Извлечение части строки, показанное как `u[1:8]`, сообщает Python, что

мы хотим извлечь все элементы набора, начиная со 2-го по 9-й. Затем используется оператор конкатенации, чтобы объединить два набора в один.

Имеется, правда, ещё более простой метод, который предоставляет Вам широкие возможности для внесения в последовательности всевозможных изменений. А состоит он в использовании списков, но это уже тема следующего раздела.

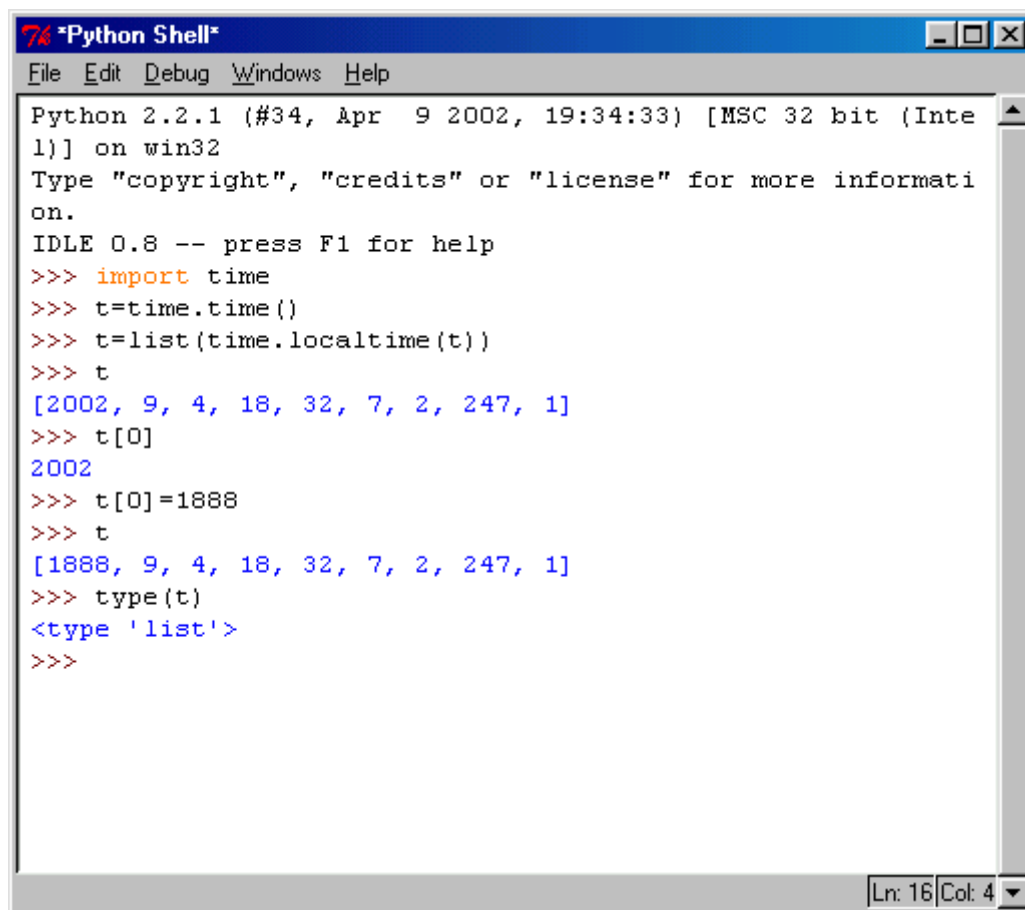
Списки

Списки во многом очень похожи на наборы, за исключением того, что они являются изменяемыми последовательностями. Кроме того, списки владеют методами. Методы почти полностью походят на функции, за исключением того, что они всегда принадлежат некоторому типу данных. Вспомните, что ранее мы уже встречались с идентификаторами модулей, когда нужно было вызвать принадлежащую им функцию. Пример такого идентификатора — вызов функции `time.time()`, где мы указываем Python, что функция `time()` принадлежит модулю `time`. Чтобы вызывать методы, необходимо также сообщать Python тип данных, которому принадлежит данный метод. Единственное отличие состоит в том, что для вызова функций модулей используются одни идентификаторы, а для вызова методов — другие.

Вы можете сообщить Python, что переменная является списком, заключив значения в квадратные скобки вместо круглых:

```
t = ["Это", "список", "из", 5, "элементов."]
e = [] # Это пустой список
```

Пример использования списка показан на рис. 6.14. Обратите внимание на отличие между наборами и списками.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following code and output:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> import time
>>> t=time.time()
>>> t=list(time.localtime(t))
>>> t
[2002, 9, 4, 18, 32, 7, 2, 247, 1]
>>> t[0]
2002
>>> t[0]=1888
>>> t
[1888, 9, 4, 18, 32, 7, 2, 247, 1]
>>> type(t)
<type 'list'>
>>>
```

The status bar at the bottom right shows "Ln: 16 | Col: 4".

Рис. 6.14. Использование списка

Как видите, изменить список намного проще, чем набор. Также легко преобразовать набор в список, просто воспользовавшись функцией `list()`. Списки, в силу их изменяемости и функциональности, часто являются самым предпочтительным типом данных. Наборы удобно применять в тех случаях, когда изменение данных не предполагается и их следует защитить от случайного изменения. Кроме того, наборы в Python занимают меньше памяти. Для небольших программ, безусловно, потребление памяти не является критичным фактором. Тем не менее всегда следует стремиться к достижению максимальной эффективности, что окажется для Вас полезной привычкой во время работы над действительно большим проектом.

Списки, как я уже говорил, содержат методы. Вот они-то как раз и определяют функциональность списков, что делает их чрезвычайно привлекательными средствами программирования. В табл. 6.4 приведены все методы, доступные для использования со списками.

Таблица 6.4. Методы, применяемые к спискам

Метод	Описание	Пример
-------	----------	--------

<code>append(s)</code>	Добавляет <i>s</i> в конец списка	<code>l.append("элемент")</code>
<code>count(s)</code>	Подсчитывает, сколько раз в списке встречается <i>s</i>	<code>l.count("элемент")</code>
<code>index(s)</code>	Определяет в списке индекс элемента <i>s</i>	<code>l.index("элемент")</code>
<code>insert(i,s)</code>	Вставляет в список <i>s</i> по месту, указанному индексом <i>i</i>	<code>l.insert(2,"элемент")</code>
<code>remove(s)</code>	Удаляет из списка первый встреченный экземпляр <i>s</i>	<code>l.remove("элемент")</code>
<code>reverse()</code>	Изменяет порядок следования элементов в списке на противоположный	<code>l.reverse()</code>
<code>sort([имя_функции])</code>	Сортирует список в порядке возрастания. Необязательный аргумент <i>имя функции</i> является именем функции сортировки (см. пояснения в тексте)	<code>l.sortd</code>

Хотя обычно в методе `sort()` нет необходимости использовать аргументы, иногда бывает полезно иметь возможность управлять порядком сортировки элементов списка. Именно это и позволяет делать функция сортировки, передающаяся в метод `sort()` в качестве аргумента. Функция сортировки в свою очередь принимает два аргумента и вызывается каждый раз при обращении к методу `sort()`. Упомянутые два аргумента функции представляют собой элементы списка. Вы сообщаете Python, в каком порядке следует сортировать список по значению, возвращаемому функцией сортировки (0, 1 или -1). По умолчанию задается сортировка по возрастанию. Выполнить сортировку по убыванию столь же просто:

```
l=["d", "e", "f", "c", "b", "a"]
l.sort()
l.reverse()
```

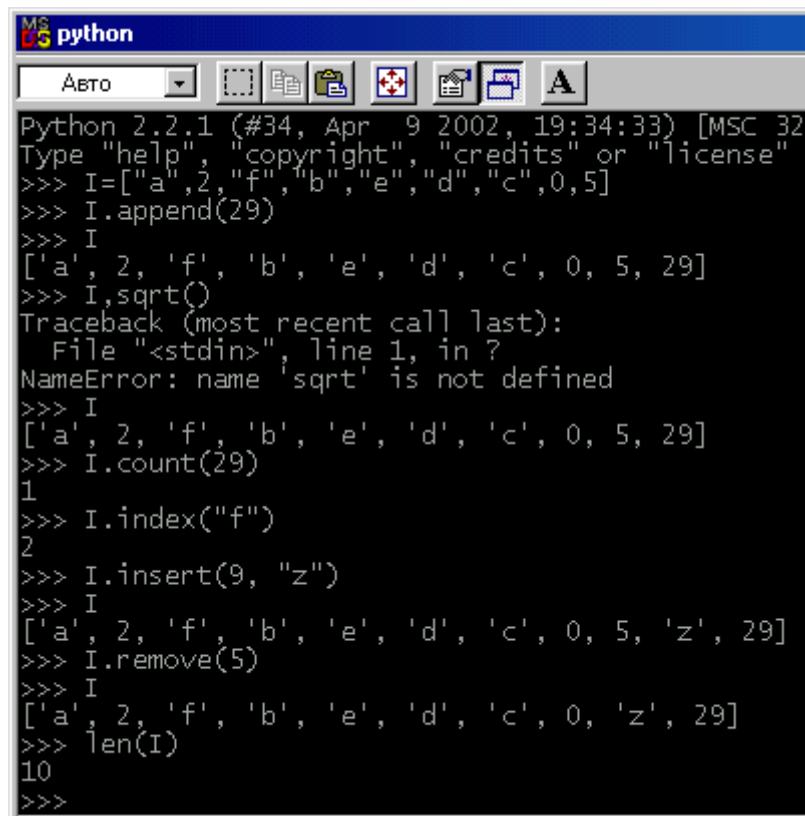
Так что для сортировки в порядке возрастания или убывания нет никакой необходимости создавать свою собственную функцию сортировки. Пример использования специальной функций сортировки показан в листинге 6.4, где функция `sort()` располагает все числа после строковых значений. Таким образом, функции сортировки необходимы тогда, когда требуемое упорядочивание элементов списка нельзя свести к сортировке по возрастанию или убыванию.

Листинг 6.4. Программа `sort.py`

```
#!c:\python\python.exe
l = ["a", 2, "f", "b", "e", "d", "c", 0, 5]
def srt(a, b):
    if type(a) == type(0) and type(b) == type(0):
        if a < b:
            return -1
        elif a > b:
            return 1
        return 0
    if type(a) == type("") and type(b) == type(""):
        if a < b:
            return -1
        elif a > b:
            return 1
        return 0
    if type(a) == type(0) and type(b) == type(""):
        return 1
    if type(a) == type("") and type(b) == type(0):
        return -1
    return 0

print l
l.sort()
print l
l.reverse()
print l
l.sort(srt)
print l
```

На рис. 6.15 показано ещё несколько методов, которыми можно воспользоваться при работе со списками.

A screenshot of a Windows-style application window titled "python". The window has a menu bar with "АВТО" and a toolbar with icons for file operations and text formatting. The main area is a black terminal window with white text showing a Python 2.2.1 shell session. The session starts with the Python version and build information, followed by a list of help topics. The user enters several commands to create and manipulate a list 'I'. The list starts with ['a', 2, 'f', 'b', 'e', 'd', 'c', 0, 5]. The user appends 29, then attempts to call 'sqrt()' on the list, which results in a 'NameError: name 'sqrt' is not defined'. The user then counts the occurrences of 29 (result: 1), finds the index of 'f' (result: 2), inserts 'z' at index 9, removes the element at index 5, and finally checks the length of the list (result: 10).

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32
Type "help", "copyright", "credits" or "license"
>>> I=['a',2,"f","b","e","d","c",0,5]
>>> I.append(29)
>>> I
['a', 2, 'f', 'b', 'e', 'd', 'c', 0, 5, 29]
>>> I.sqrt()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'sqrt' is not defined
>>> I
['a', 2, 'f', 'b', 'e', 'd', 'c', 0, 5, 29]
>>> I.count(29)
1
>>> I.index("f")
2
>>> I.insert(9, "z")
>>> I
['a', 2, 'f', 'b', 'e', 'd', 'c', 0, 5, 'z', 29]
>>> I.remove(5)
>>> I
['a', 2, 'f', 'b', 'e', 'd', 'c', 0, 'z', 29]
>>> len(I)
10
>>>
```

Рис. 6.15, Методы обработки списков

Списки — это последний тип данных, относящийся к последовательностям. Следующий раздел посвящен словарям, которые существенно отличаются от ранее рассмотренных типов, но являются чрезвычайно полезными и удобными в программах на языке Python.

Словари

Словари в какой-то мере подобны последовательностям, только их элементы не хранятся в каком-либо определенном порядке. Они представляют собой совокупности пар ключ—значение. Вы помещаете информацию в словарь, предоставляя Python ключ (в качестве которого может выступать почти любой допустимый в Python тип данных), с которым сопоставляется значение, которое, в свою очередь, также может быть представлено данными любого типа. Наиболее распространенным типом ключа являются строки, но допускается использовать и целые числа, длинные целые числа и наборы. В качестве ключей нельзя применять только списки и словари. Ключ является неизменяемым элементом словаря, поэтому удобно в качестве ключей использовать константные, или вычисляемые значения, которые затем преобразуются в константные типы данных. Цифры, используемые в программном коде, являются константами, поскольку выражения типа $1 = 2$ воспринимаются Python как ошибки.

Вот способ создания пустого словаря:

```
diet = {}
```

Признаком словаря в Python служат фигурные скобки, даже если между ними ничего не заключено. Присвоить пустому словарю первый элемент, состоящий из значения и ключа, можно следующим образом:

```
dict["myri"] = 3.14159
```

В данном примере ключом будет строка `myri`, а значением — число с плавающей запятой `3.14159`. Чтобы извлечь значение, в качестве индекса просто применяется ключ, связанный с этим значением:

```
myri = dict["myri"]
```

Если указанный ключ в момент вызова ещё не существует в словаре, Python покажет сообщение об ошибке. Для создания в одной строке полноценного словаря с ключами и значениями используется следующий синтаксис:

```
diet={"first":1, "second":2, "third":3, "eleventh":11}
```

Хотя Вы можете задавать элементы словаря в определённом порядке, Python будет сохранять ключи и значения в произвольном, только ему известном порядке. Когда я ввёл предшествующую строку, а затем распечатал переменную `diet`, то вот что я получил:

```
{'first': 1, 'third': 3, 'eleventh': 11, 'second': 2}
```

***Прим. В. Шипкова: в Python 2.4.1 список выводится с точностью до наоборот.
{'second': 2, 'eleventh': 11, 'third': 3, 'first': 1}**

Нет смысла заставлять Python хранить элементы словаря в каком-либо порядке. Если необходимо вывести содержимое словаря в определённой последовательности, используйте встроенный метод `keys()`, который отсортирует элементы словаря по ключам, как это показано на рис. 6.16.

```

Python 2.2.1 (#34, Apr  9 2002, 19:34:33) [MSC 32 bit (Intel)
] on win32
Type "copyright", "credits" or "license" for more information
.
IDLE 0.8 -- press F1 for help
>>> dict={"first":1, "second":2, "third":3, "eleventh":11}
>>> k=dict.keys()
>>> k
['second', 'eleventh', 'third', 'first']
>>> k.sort()
>>> k
['eleventh', 'first', 'second', 'third']
>>> |

```

Рис. 6.16. Сортировка элементов словаря по ключам

Для работы со словарями применима большая часть операторов обработки последовательностей, за исключением операторов извлечения подстрок. Множество других полезных функций в работе над словарями можно реализовать с помощью методов. В табл. 6.5 приведён список всех методов, доступных для обработки словарей.

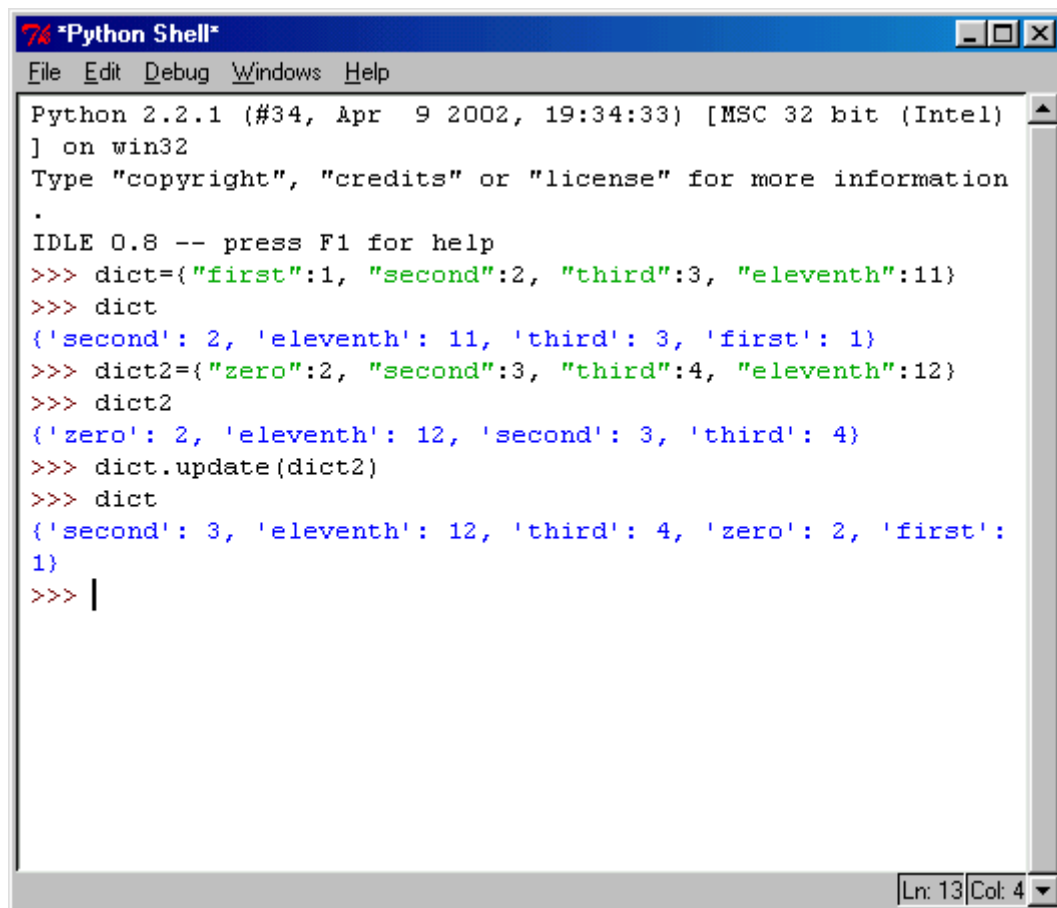
Таблица 6.5. Методы обработки словарей

Метод	Описание	Пример
<code>clear()</code>	Удаляет из словаря все элементы	<code>dict.clear()</code>
<code>copy()</code>	Возвращает экземпляр (копию) всех элементов словаря	<code>newcopy=dict. copy()</code>
<code>get(ключ[, по умолчанию])</code>	ещё один метод возвращения значения по ключу, но в этом случае,	<code>x=dict.get("Неправильный ключ")</code>

	если указанный ключ в словаре не обнаружен, метод возвращает None или значение, заданное по умолчанию	
<code>haskey(k)</code>	Возвращает 0, если данный ключ не существует, иначе -1	<code>t=dict.haskey("Spam")</code>

<code>items()</code>	Возвращает список наборов, состоящих из пар значений (ключ, значение) для всех элементов словаря	<code>t=dict.items()</code>
<code>keys()</code>	Возвращает список всех ключей словаря	<code>l=dict.keys()</code>
<code>update(dict2)</code>	Обновляет словарь по образцу <i>dict2</i> (см. пояснения в тексте)	<code>dict.update(dict2)</code>
<code>values()</code>	Возвращает список всех значений, содержащихся в словаре	<code>v=dict.values()</code>

Большинство приведенных методов вполне очевидны и понятны, за исключением метода `update()`, пример использования которого показан на рис. 6.17.

A screenshot of a Python Shell window titled "Python Shell". The window has a menu bar with "File", "Edit", "Debug", "Windows", and "Help". The main text area shows the following code and output:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information
.
IDLE 0.8 -- press F1 for help
>>> dict={"first":1, "second":2, "third":3, "eleventh":11}
>>> dict
{'second': 2, 'eleventh': 11, 'third': 3, 'first': 1}
>>> dict2={"zero":2, "second":3, "third":4, "eleventh":12}
>>> dict2
{'zero': 2, 'eleventh': 12, 'second': 3, 'third': 4}
>>> dict.update(dict2)
>>> dict
{'second': 3, 'eleventh': 12, 'third': 4, 'zero': 2, 'first': 1}
>>> |
```

The status bar at the bottom right shows "Ln: 13 Col: 4".

Рис. 6.17. Обновление словаря

Словари можно использовать для хранения информации любого типа, даже других словарей. Это свойство присуще также таким типам данных, как набор и список. Все они могут хранить элементы любого типа, включая данные того же самого типа, т.е. другие вложенные наборы, списки и словари. Словари особенно удобны в тех случаях, когда порядок следования элементов не имеет значения, но есть необходимость позволить пользователям извлекать данные по определенным ключам, которые можно вводить, например, с клавиатуры. Давайте рассмотрим такой небольшой прикладной пример. За последние 400 лет изучения календаря индейцев Майя разными учеными было предложено множество терминов для обозначения месяцев в этом календаре. Недавно были приняты стандартные названия, каждому из которых соответствует множество синонимов. Чтобы дать возможность пользователям моей программы удостовериться, соответствует ли выбранное ими название более привычному для них синониму, я разработал словарь. В этом словаре ключами являются всевозможные варианты названий месяцев календаря Майя (строки), тогда как значениями выступают порядковые номера этих месяцев в году (целые числа). Это значительно упростило пользовательский интерфейс по сравнению с аналогичной версией, которую я написал на языке С много лет

назад, а также повысило эффективность и гибкость интерфейса.

Код словаря месяцев Майя показан в листинге 6.5 (в каждом списке новые стандартные названия перечислены первыми).

```
monthnames = {
    "pohp":0, "pop":0, "kanhalaw":0, "k'anhalaw":0,

    "wo":1, "uo":1, "ik'k'at":1, "ikk'at":1,
    "ik'kat":1, "ikkat":1,

    "sip":2, "zip":2, "chakk'at":2, "chakkat":2,

    "sots":3, "zots":3, "zotz":3, "tsots":3,
    1 "tzots":3, "suts":3, "suts'":3, "sutz":3, "sutz'":3,

    1 "sek":4, "sec":4, "zec":4, "zek":4, "tsek":4,
    1 "tsek":4, "kasew":4,

    1 "xul":5, "chichin":5,

    "yaxk'in":6, "yaxkin":6,

    "mol":7,

    "ch'en":8, "chen":8, "ik'":8, "ik":8,

    "yax":9,

    "sak":10, "zak":10, "tsak":10, "tzak":10,

    "keh":11, "ceh":11, "chak":11,

    "mak":12, "mac":12,

    "k'ank'in":13, "kank'in":13, "kankin":13, "uniw":13,

    "muwan":14, "muan":14, "moan":14,

    "pax":15, "ah k'ik'u":15, "ahk'ik'u":15, "ahkik'u":15,
    "ah kik'u":15, "ahk'iku":15, "ah k'iku":15,
    "ahkiku":15,
    "ah kiku":15,

    "k'ayab":16, "kayab":16, "cayab":16, "kanasi":16,
    "k'anasi":16,
```

```
"kumk'u":17, "kumku":17, "cumku":17, "cumhu":17,  
"cum'hu":17, "ol":17,  
  
"wayeb":18, "uayeb":18  
}
```

Словари очень удобны также при формировании так называемой *таблицы распределений*, в которой по ключу определяется функция, которую необходимо вызвать. Например, в рассмотренном выше словаре месяцев Майя можно легко заменить числовые значения номера месяца на имена функций, чтобы при вводе пользователем одного из синонимов названия месяца каждый раз вызывалась соответствующая функция. Можно изменить словарь таким образом, чтобы функции вызывались только при вводе нестандартных названий месяцев и автоматически заменяли их на стандартные.

Другой благодатной почвой применения словарей является обработка выбора опций меню. Предположим, что Вы хотите предоставить пользователю на выбор множество опций. Основываясь на обычных приёмах, Вам пришлось бы создать сложную конструкцию из условных операторов, как в следующем примере:

```
if выбор_пользователя == 1:  
    функция1()  
elif выбор_пользователя == 2:  
    function2()
```

Но благодаря применению словарей эта процедура становится значительно проще:

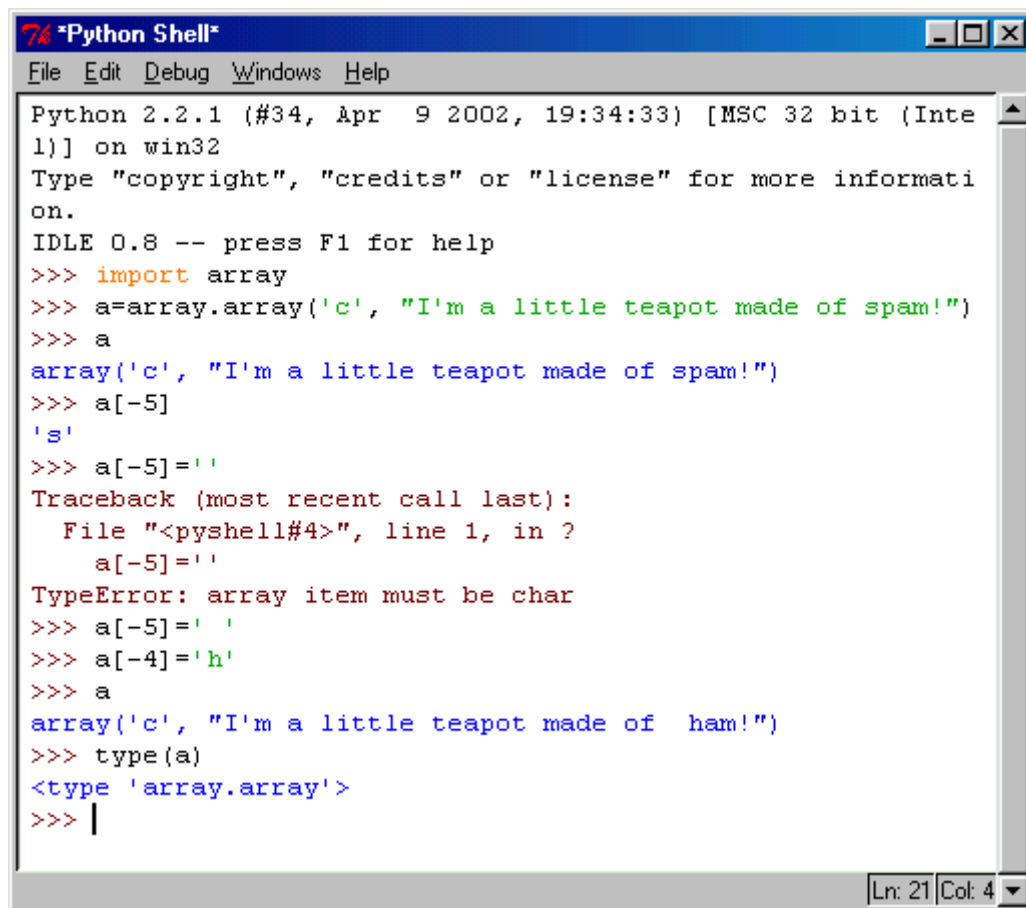
```
diet={" 1 ":функция1, " 2 ":функция2, ...}  
... возвращение выбора пользователя любым способом —  
dict[user_choice] ( )
```

Массивы

Python обладает огромным количеством модулей расширения. Документация на все модули, как стандартные так и расширения, может быть найдена на уже известном Вам сервере <http://www.Python.org/>. Во время установки Python я настоятельно рекомендую наряду с Python выгрузить и установить документацию в формате HTML. Иметь этот справочник всегда под рукой значительно удобнее, чем заходить каждый раз в Internet, особенно если у Вас медленная связь с провайдером и Вы оплачиваете время работы в сети. Многим из тех, кто переходит к языку Python от

других языков программирования, поможет адаптироваться модуль `array`, который предоставляет средства обработки массивов, аналогичные языку C. Массивы во многом напоминают списки и наборы. Это изменяемый тип данных, но на их использование наложено ограничение, требующее, чтобы массивы содержали только однородные данные. Примеры использования массивов показаны на рис. 6.18.

Как Вы видите, элементы массивов можно легко изменять. Массивы символов можно преобразовать в строки символов, а также выполнять обратные преобразования с помощью метода `fromstring()`. В любой момент, когда Вам понадобится считать или записать двоичную информацию из файла или в файл, в Вашем распоряжении имеются превосходные методы обработки массивов. Такие задачи (а они не из самых приятных) рано или поздно приходится решать каждому программисту. Более подробную информацию можно найти в документации на массивы. ещё раз обратите внимание на один важный момент: массивы могут содержать элементы только одного типа. Это объясняется тем, что все элементы массива должны иметь один и тот же размер. Но те же ограничения иногда возникают при совмещении модулей, написанных на разных языках программирования, или при передаче данных по сети. Например, для корректной передачи данных по сети часто бывает необходимо предварительно привести все значения к шестнадцатеричному формату. В этом случае использование массивов усилит Ваш контроль за вводимыми данными.

The image shows a screenshot of a 'Python Shell' window. The title bar says 'Python Shell' with standard window controls. The menu bar includes 'File', 'Edit', 'Debug', 'Windows', and 'Help'. The main text area shows the following interaction:

```
Python 2.2.1 (#34, Apr 9 2002, 19:34:33) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>> import array
>>> a=array.array('c', "I'm a little teapot made of spam!")
>>> a
array('c', "I'm a little teapot made of spam!")
>>> a[-5]
's'
>>> a[-5]=''
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in ?
    a[-5]=''
TypeError: array item must be char
>>> a[-5]=' '
>>> a[-4]='h'
>>> a
array('c', "I'm a little teapot made of ham!")
>>> type(a)
<type 'array.array'>
>>> |
```

The status bar at the bottom right shows 'Ln: 21 | Col: 4'.

Рис. 6.18. Использование массивов

Резюме

Изучая материал данной главы, нам пришлось изрядно потрудиться. Вы познакомились с использованием строк символов, включая использование функций модуля `string`. Вы узнали, чем наборы отличаются от списков, и научились применять методы обработки списков, а также познакомились со словарями и их методами. Далее мы будем часто обращаться в своих программах к словарям. И наконец, мы обсудили с Вами особенности применения массивов. В следующей главе Вы научитесь определять функции. А помогут Вам в этом знания базовых типов данных, которыми Вы уже овладели.

Практикум

Вопросы и ответы

Почему строки символов Python являются неизменяемым типом данных?

Это снизило бы эффективность работы программ на языке Python. Не сложно было сделать строки символов изменяемыми, но это сильно замедлило бы работу, поскольку управление выделением памяти для изменяемых переменных происходит

значительно медленнее. На практике строки используются очень часто, а необходимость в их изменении возникает довольно редко.

Используют ли словари другие языки?

Да. Их использует SNOBOL, который является языком обработки строк символов. Две программы оболочки командной строки под UNIX, bash и ksh, также используют словари, хотя в этом случае они называются ассоциативными массивами. Словари интенсивно использует ещё один довольно скромный язык программирования — AWK, разработанный для систем UNIX. Применяются словари и в Perl, причём одним из его прототипов служил язык AWK, и во многих других языках программирования.

В каком случае все же удобно использовать наборы?

Это, безусловно, полезное средство программирования. Но учитывая небольшой размер и ограниченную функциональность большинства программ, с которыми Вы будете работать по ходу чтения этой книги, наборы практически всегда будут уступать по эффективности спискам. Преимущество наборов можно оценить только во время работы над большими проектами, когда критичным становится расход памяти компьютера. Наборы, в отличие от списков, гораздо экономичнее в смысле потребления памяти и других ресурсов. Кроме того, многие функции возвращают наборы, а не списки.

Контрольные вопросы

1. Можно ли в операторах for использовать строки символов?
 - а) Нет.
 - б) Да, но сначала необходимо обработать их с помощью функции, преобразующей их в последовательности.
 - в) Да, так как строки символов уже являются последовательностями.
 - г) Только если строки содержат последовательности цифр.
2. Что в словарях может быть использовано в качестве ключей?
 - а) Целые числа и длинные целые числа.
 - б) Строки символов и наборы.
 - в) Числа с плавающей запятой и комплексные числа.
 - г) Всё вышесказанное.
3. Что означает оператор + применительно к строкам?

- а) Конкатенацию двух строк символов в одну новую.
- б) Суммирование кодов ASCII всех символов обеих строк.
- в) Прежде чем использовать этот оператор, его нужно предварительно переопределить в программе.
- г) Объединение двух строк в одну со знаком ' + ' между ними.

4. Что такое идентификаторы?

- а) Средства проверки уровня безопасности ваших программ.
- б) Функции, которые используют для преобразования массивов символов в строки.
- в) Средство указания полного имени функции, включая имя модуля. Применение идентификаторов позволяет избежать возникновения конфликтов имён.
- г) Средства, обеспечивающие переносимость программ на языке Python между разными платформами.

Ответы

- 1. в. Строки символов являются последовательностями, поэтому вполне допустимо (а часто и полезно) их использование в операторах for.
- 2. г. В словарях в качестве ключей можно использовать все указанные типы данных. Нельзя использовать только списки, словари и объекты, определяемые пользователем.
- 3. а. Оператор + выполняет конкатенацию двух строк для создания новой строки.
- 4. в. Идентификаторы используют для указания полного имени функции, включая имя модуля, что позволяет избежать возникновения конфликтов имён.

Примеры и упражнения

Чтобы лучше подготовиться к темам, которые мы будем рассматриваться в следующих главах, ещё раз просмотрите код программы lbm.py (листинг 6.3) и проверьте, сможете ли Вы объяснить, что именно выполняет каждая строка кода, в том числе и те из них, которые мы не рассматривали. В случае возникновения затруднений внимательно пересмотрите документацию на Python, выгружаемую из Internet в формате HTML. Сможете ли Вы переписать программу lbm.py так, чтобы она состояла из меньшего количества строк кода? Сокращать код — один из лучших приёмов научиться программированию. Естественно, что сокращение кода не должно сопровождаться потерей функциональности программы. Вам часто на практике придётся выполнять эту задачу. Исходные коды программы обычно оказываются неоправданно длинными из-за того, что



программист сохраняет на всякий случай свои промежуточные наработки, которые затем нигде не используются и лишь захламляют код. И это в лучшем случае. Часто в этом хламе заводятся "жучки", которые вызывают ошибки в работе программы и затрудняют их поиск во время отладки. Поэтому имеет смысл время от времени "прочёсывать" весь код и избавляться от всякой чепухи, в которой Вы больше не нуждаетесь.

7-й час

Функции и модули

В предыдущей главе Вы узнали о таких базовых типах данных Python, как последовательности и словари. Таким образом, Вы уже готовы к написанию функций. Только после того как Вы овладеете искусством написания функций, можно переходить к созданию эффективных и полезных программ. Мы также рассмотрим модули, упоминания о которых уже встречались Вам раньше. Модули — это файлы, в которых собраны определения функций. К концу этой главы Вы не „только научитесь создавать функции, но и узнаете, как помещать функции в модули, чтобы их можно было импортировать и использовать в других программах. Итак, мы рассмотрим следующие вопросы: определение собственной функции; создание собственного модуля; проверка имени модуля; использование аргументов командной строки.

Для чего нужны функции

Единственное назначение функции состоит в том, чтобы хоть немного облегчить труд программиста. Без функций код программы превращается в длинное "спагетти", связанное в невообразимые узлы инструкциями `goto` и `long_jump`. Разобраться в таких хитросплетениях было невозможно ещё на заре программирования, когда программы по сложности и обилию выполняемых задач были несоизмеримо проще современных.

Функция `long_jump()` печально известна в системах UNIX. С её помощью можно выполнять так называемые нелокальные переходы. Такие переходы позволяют программистам, работающим в C на машинах с UNIX, произвольно перескакивать из одного блока кода в другой, независимо от текущего состояния программы. Когда их применение обосновано и выполнено должным образом, это может стать верхом искусства, но стоит чуть-чуть расслабиться, результаты окажутся столь же живописными, как прыжок ребенка в лужу грязи.

Наша задача состоит в том, чтобы написанные программы были просты для понимания и читабельны. Создание ясного и понятного кода — вот одна из причин, почему существуют функции. С их помощью Вы можете разбивать длинные уродливые нагромождения тяжело воспринимаемого кода в небольшие и лёгкие для понимания блоки. Кроме того, эти блоки можно использовать в программе многократно. Многократное использование функций — очень важное свойство, впрочем, присущее всем языкам программирования. Но Python отличается тем, что значительно облегчает процесс многократного использования одних и тех же функций в независимых программах. Нет никакого здравого смысла в многократном переписывании одного и того же кода для каждой создаваемой Вами программы.

Разбиение программы на множество небольших блоков (тут, конечно, тоже не следует перегибать палку) значительно повышает эффективность программирования, а не просто облегчает восприятие и понимание разных блоков кода. Даже возможность повторного использования функций уходит на второй план по сравнению с тем, что само по себе использование функций в корне изменяет стиль и основополагающие принципы программирования. Для наглядности позвольте мне рассказать об одной из своих самых первых программ, написанных в далеком 1968 году (примите во внимание, что это была эпоха повального увлечения кодом в стиле "спагетти").

В то время я работал оператором вычислительной машины на третьей смене. Каждую неделю мы должны были усаживаться за клавишный перфоратор (а как ещё, по-вашему, появлялись те квадратные дырочки в перфокартах) и набивать титульные перфокарты для хранилища магнитных лент с программами. Эти перфокарты затем загружали в специальную инвентаризационную машину. Каждую неделю нужно было набить приблизительно 50 карточек, что занимало час или два усердной работы, в зависимости от того, насколько проворно мы работали за перфоратором. Вскоре после того как я познакомился с основами программирования на COBOL, я понял, что мог бы заставить компьютер вместо меня перфорировать карты, вставляя текущую дату. Все, что мне необходимо было сделать, — это написать программу, которая бы выполняла эту задачу.

Поскольку перфокарты были необходимы только в рабочие дни, а я хотел, чтобы моя программа работала веками, мне пришлось погрузиться в особенности григорианского календаря, т.е. проблема состояла в том, чтобы программа отслеживала рабочие дни в разные годы. Я выяснил, что годы

полностью повторяются с периодичностью 28 лет, с незначительными исключениями, проявляющимися каждые 100 лет. Чтобы внести свою лепту в проблему 2000 года, я решил проигнорировать эти незначительные исключения, которые, конечно же, означали, что я использовал юлианский календарь, а не григорианский.

Это происходило в то время, когда до восхода структурного программирования ещё было далеко. Не будучи гением, я не размышлял о всяких там функциях с параметрами, поэтому делал то, что в то время делал бы любой другой, — использовал бесчисленное множество инструкций перехода `goto`. Сначала я вычислил положение текущего года в 28-летнем цикле, а затем сделал переход к довольно массивному блоку, вычисляющему рабочие дни для годов данной категории. Таким образом, у меня в программе присутствовало 28 почти идентичных блоков кода. Если я находил ошибку в одном месте, то мне приходилось исправлять её 28 раз. В настоящее время аналогичную операцию можно было бы реализовать на Python в виде одной функции, занимающей всего десяток строк кода, заменив все те тысячи строк невероятно многословного языка COBOL. Моя программа работала, но до проблемы 2000 года не дожила. Прогресс в компьютерной технике очень скоро сделал её ненужной.

Комбинация из инструкций `goto`, нескольких строк кода и другой `goto`, которая осуществляет переход назад на строку, следующую за исходной `goto`, когда-то имела особое название — *подпрограмма*. Если взглянуть на код ассемблера, производимый современными трансляторами, то можно заметить огромное сходство между кодом, сгенерированным для реализации какой-нибудь функции, и логическим построением подпрограммы, которые мы обычно использовали в те времена. Да, это верно, что по сути своей функции являются всего лишь только красивой оберткой, за которой скрывается старая фундаментальная конструкция — инструкция `goto`, код подпрограммы и другая инструкция `goto`. Но именно эта блестящая обёртка позволяет нам мыслить более логично и предельно упрощать код программ. А как мы помним из главы 1, структура языка определяет ход мышления. И эта закономерность особенно ярко проявляется в языках программирования.

Давайте рассмотрим очень маленькую функцию, с элементами которой Вы уже встречались прежде. В листинге 7.1 показано, как выполняется определение функции в Python. Где-то мы такое уже видели, не так ли?

Листинг 7.1. Определение функции

```
def julian_leap(y):  
    if (y%4) == 0:  
        return 1  
    return 0
```

Ключевое слово `def` является инструкцией определения функции. За ним следует произвольное имя функции, круглые скобки, в которых содержится список параметров, и завершает строку двоеточие. Тело функции выделяется отступом от строки, содержащей `def`. Имена, заключённые между круглыми скобками, являются аргументами функции или параметрами, и для них можно использовать произвольные имена. Они являются всего лишь только удобными идентификаторами, указывающими, какую именно информацию следует передавать функции. Область видимости аргументов ограничена пределами тела функции. Что это означает? Например, в нашем случае для функции была создана временная переменная под именем `y`. Эта переменная существует только внутри этой функции. Переменные с таким же именем можно использовать в любой другой части кода, даже если перед этим уже происходил вызов функции `julian_leap()`.

В листинге 7.2 показано, как можно использовать функцию, созданную в предыдущем примере.

Листинг 7.2. Пример вызова функции

```
year = 1999  
if julian_leap(year):  
    print year, "is leap"  
else:  
    print year, "is not leap"
```

Теперь Вы можете убедиться, что если заменить, вызов функции инструкцией `goto` (правда, в этом случае это уже был бы не Python), а также ввести инструкции возврата с другой инструкцией `goto`, то мы получим обычную подпрограмму. Инструкцию `return` можно рассматривать как своего рода команду "вернуться обратно". Компилятор или интерпретатор фиксирует место, из которого вызывается функция, поэтому инструкция `return` означает просто "перейти в то место программы, которое следует непосредственно за вызовом функции". Вообще-то на низком уровне (уровне команд процессора) все конструкции управления ходом выполнения программ, которые мы уже изучили, являются всего лишь только наборами команд проверки исходных адресов и переходов. Даже цикл `for` транслируется интерпретатором Python в набор команд `goto`, сгруппированных вместе с

командами проверки текущих значений переменных, контролирующих окончание списка входных данных. Но насколько логичнее становится код программы, когда вместо бесконечных скачков, переходов и проверок в ней появляются простые и удобочитаемые инструкции `if`, `for`, `while`, а также функции.

Наша функция `julian_leap()` ограничена использованием юлианского календаря, так как в ней предполагается, что високосным является каждый четвёртый год. Несложно изменить эту функцию для приведения в соответствие с григорианским календарём. В листинге 7.3 демонстрируется этот код.

Листинг 7.3. Функция определения високосного года по григорианскому календарю

```
def gregorian_leap(y):  
    if (y%400) == 0:  
        return 1  
    elif (y%100) == 0:  
        return 0  
    elif (y%4) == 0:  
        return 1  
    return 0
```

Все это Вы уже видели прежде, но теперь код упакован в отдельную функцию. В листинге 7.4 показан вызов обеих функций.

Листинг 7.4. Вызов функций определения високосного года по юлианскому и григорианскому календарям

```
years = [1999,2000,2001,1900]  
for x in years :  
    if julian_leap(x):  
        print "Julian", x, "is leap"  
    else:  
        print "Julian", x, "is not leap"  
    if gregorian_leap(x):  
        print "Gregorian", x, "is leap"  
    else:  
        print "Gregorian", x, "is not leap"
```

Переменная `years` содержит список годов (имеется в виду не просто список, а такой тип данных, о котором, я надеюсь, Вы ещё помните), представленных целыми числами. Цикл `for` перебирает все элементы этого списка и выполняет для каждого значения года обе функции `julian_leap()` и

`gregorian_leap()`, после чего на экран выводятся результаты их выполнения.

Python допускает назначение аргументам функции значений, задаваемых по умолчанию. Делается это очень просто, гораздо проще, чем в других языках программирования. Заданные по умолчанию значения аргумента используются в том случае, если при вызове функции соответствующий аргумент не был определён. Давайте попробуем изменить функцию `julian_leap()` таким образом, чтобы предоставить ей значение по умолчанию, как показано в листинге 7.5.

Листинг 7.5. Аргументы, заданные по умолчанию

```
def julian_leap(y=2000):  
    if (y%4) == 0:  
        return 1  
    return 0
```

Обратите внимание, что всё, что нам необходимо было сделать, — это присвоить аргументу `y` некоторое значение в определении функции. Если во время вызова функции не будет передан аргумент, то переменная `y` примет присвоенное ей по умолчанию значение 2000. Но если пользователь всё-таки сподобится снабдить вызов функции аргументом (в нашем случае — значением года), то переменной `y` будет присвоено именно это значение. Все предельно просто, не так ли? Можете проверить все вышесказанное, выполнив код, показанный в листинге 7.6.

Листинг 7.6. Вызов функции, в которой задан аргумент по умолчанию

```
if julian_leap():  
    print "Julian 2000 yes"
```

В листинге 7.7 используются те же функции, которые мы обсуждали до этого, но теперь код программы помещен в отдельный файл. Воспользуйтесь своим текстовым редактором, чтобы набрать эти строки, и сохраните их в файле под именем `leap2.py`.

Листинг 7.7. Файл `leap2.py`

```
#!/c:\python\python.exe  
import sys  
import string
```

```

def julian_leap(y=2000):
    if (y%4) == 0:
        return 1
    return 0

def gregorian_leap(y=2000):
    if (y%400) == 0:
        return 1
    elif (y%100) == 0:
        return 0
    elif (y%4) == 0:
        return 1
    return 0

years = [1999,2000,2001,1900]
print julian_leap()
print gregorian_leap()
if julian_leap():
    print "Julian 2000 yes"
if gregorian_leap():
    print "Gregorian 2000 yes"
for x in years:
    if julian_leap(x):
        print "Julian", x, "is leap"
    else:
        print "Julian", x, "is not leap"

    if gregorian_leap(x):
        print "Gregorian", x, "is leap"
    else:
        print "Gregorian", x, "is not leap"

```

Если после того как Вы сохранили текст предыдущей программы в файле под именем leap2.py, ввести в командной строке python leap2.py, то Вы должны увидеть результаты, аналогичные приведенным в листинге 7.8.

```

c:\python\python.exe leap2.py
Julian 2000 yes
Gregorian 2000 yes
Julian 1999 is not leap
Gregorian 1999 is not leap
Julian 2000 is leap
Gregorian 2000 is leap
Julian 2001 is not leap
Gregorian 2001 is not leap
Julian 1900 is leap
Gregorian 1900 is not leap

```

С аргументами функций можно проделывать множество причудливых фокусов, но, вообще говоря, в большинстве случаев Вам вряд ли понадобятся подобные уловки. Когда Вы приобретёте некоторый опыт и реализуете несколько более-менее сложных задач, то будете способны принимать более изящные решения в вопросах передачи аргументов. К тому же, очень многое на эту тему можно почерпнуть либо из других книг, либо из официальной документации, которую можно загрузить вместе с установочной программой Python. Сейчас Вы уже обладаете достаточными знаниями о функциях, чтобы на 90% удовлетворить свои потребности в программировании. Поэтому давайте перейдем к следующей теме – модулям.

Модули

Всякий раз, когда в предыдущих программах мы использовали выражение `import модуль` или `from модуль import *`, в программу добавлялся соответствующий модуль. Под модулем следует понимать файл. Имя этого файла должно заканчиваться стандартным для Python расширением `.py`. Вспомните особенности уже известных Вам приёмов импортирования. Если модуль импортируется инструкцией `import модуль`, необходимо перед именем каждой функции этого модуля устанавливать идентификатор – имя самого модуля. Если же используется инструкция `from модуль import *`, то все функции модуля становятся частью программы, но тогда могут возникнуть конфликты имён с одноимёнными функциями, которые были определены в самой программе или в других импортированных модулях. (Помните, что произошло с модулями `math` и `cmath` в главе 5.) Инструкция импорта требует указания имени файла модуля, но у программы "поедет крыша", если Вы ещё укажете и расширение `.py`.

***Прим. В. Шипкова: ко всему сказанному добавлю от себя – имя модуля НЕ ДОЛЖНО начинаться с цифры, хотя файл, так может вполне обзывать. Питон будет усиленно ругаться, и это вполне логично – имя модуля Питон рассматривает как переменную, а переменная не может начинаться с цифры.**

Прежде чем Вы сможете создать свой собственный модуль функций, что позволит многократно использовать эти функции в разных программах, необходимо познакомиться с одной идиомой Python. На самом деле Вы уже сталкивались с ней прежде, но мы не обсуждали эту особенность. Каждая выполняемая программа в Python имеет своё имя. Даже когда Вы выполняете программу путём ввода её команд в окне

интерпретатора (т.е. в диалоговом режиме), программа имеет имя `"__main__"`. Если же программа выполняется в диалоговом режиме, но в окне IDLE, то она носит имя `"__console__"`.

***Прим. В. Шипкова: в Python 2.4.1
имя программы также `"__main__"`**

Это отличие может показаться несколько странным, но в действительности Вам редко придётся сталкиваться с этими именами, поскольку они создаются в Python автоматически. Можно узнать текущее имя выполняемой программы. Для этого следует вывести на экран содержимое встроенной переменной `__name__` (о встроенных переменных мы поговорим подробнее в главе 11).

В большинстве случаев выводимое значение будет `__main__`. Обратите внимание, что `__name__` не является именем модуля или другого файла.

Откройте свой излюбленный текстовый редактор и введите код, показанный в листинге 7.9, после чего сохраните свою работу в файле под именем `name.py`.

Листинг 7.9. Программа `name.py`

```
#! c:\python\python.exe
import leap2
print __name__
```

Удостоверьтесь в том, что эта программа находится в том же самом каталоге, в котором Вы сохранили `leap2.py`, а затем запустите её обычным способом — вводом команды `python name.py`. Ожидаемый результат показан в листинге 7.10.

Листинг 7.10. Результаты выполнения программы `name.py`

`C:\Python>python name.py`

```
Julian 2000 yes
Gregorian 2000 yes
Julian 1999 is not leap
Gregorian 1999 is not leap
Julian 2000 is leap
Gregorian 2000 is leap
Julian 2001 is not leap
Gregorian 2001 is not leap
Julian 1900 is leap
Gregorian 1900 is not leap
```


Что произошло? Дело в том, что когда Python выполнил инструкцию импорта, он также выполнил и весь код, заключенный внутри модуля `leap2`. Если изменить файл `leap2.py`, включив в него (как показано в листинге 7.11) проверку имени программы (убедитесь только, что должным образом установлен отступ строк, следующих за инструкцией `if`), то тем самым Вы сможете предотвратить нежелательное выполнение модуля.

Листинг 7.11. Проверка имени, добавленного в файл `leap2.py`

```
if __name__ == "__main__":  
    years = [1999, 2000, 2001, 1900]  
    .....
```

Импортированные модули тоже имеют свою переменную `__name__`, которой присваивается имя файла. Так, в модуле `leap2`, который хранится в файле `leap2.py`, имеется переменная `__name__`, значение которой установлено в `"leap2"`. Но если запустить файл `leap2` на выполнение самостоятельно, а не как модуль, то в переменной `__name__` будет сохранено имя `"__main__"`. Обычно имя модуля, присвоенное переменной `__name__`, мало кого интересует. Уж если Вы смогли импортировать модуль, то наверняка знаете его имя. Но проверка соответствия переменной `__name__` имени `"__main__"` имеет свой смысл. Это позволяет нам отменять автоматическое выполнение программы, если она загружается как модуль в другую программу. Проверьте это ещё раз, выполнив измененный файл `leap2.py`, в котором Вы добавили проверку имени модуля. Результат должен получиться тот же самый, что и в первый раз. Кроме того, теперь Вы можете использовать функции определения високосного года в любой программе, в которую импортируете модуль `leap2`. Причем программа самого модуля выполняться не будет. Так, если теперь Вы выполните программу `name.py`, на экране появится только имя текущего модуля `__main__`.

Чтобы позволить модулю `__main__` взаимодействовать с пользователями, необходимо снабдить его возможностью считывать параметры командной строки, которые вводит пользователь при запуске программы на исполнение. Для этого в основной модуль программы необходимо добавить строку `import sys`. Всякий раз, когда программа будет запущена на выполнение, интерпретатор Python будет собирать все параметры командной строки и помещать их в специальный список `argv` в модуле `sys`.



Список `argv` расшифровывается как вектор аргументов (*argument vector*). Назван он так по аналогии с соответствующим вектором, используемым в языке *C*, хотя в *Python* это список. Вектор — это особый вид массива.

Кстати, имя `__main__` также пришло из языка *C*. Программы на этом языке всегда имеют главную функцию с именем `main()`, с которой начинается их выполнение. В *Python*, как и в *C*, самому первому элементу вектора аргументов `argv[0]` всегда присваивается имя программы.

***Прим. В. Шипкова: вообще Python это термоядерная смесь приличного количества языков.**

Поскольку `argv` является списком, обратиться к нему очень просто. С помощью встроенной функции `len()` можно сначала выяснить, сколько параметров ввёл пользователь:

```
l=len(sys.argv)
```

Как было указано в приведённом выше примечании, первый элемент списка `argv[0]` всегда содержит имя программы (имя файла, которое мы вводим при запуске программы). Это полезное свойство, так как в диалоговых окнах, экранных примечаниях и сообщениях об ошибках легко можно выводить имя программы, обратившись к `sys.argv[0]`, вместо того чтобы жёстко вводить имя в код сообщений. Если размер списка `sys.argv` равен единице, это значит, что пользователь не вводил никакие параметры. Можно управлять выполнением программы в зависимости от того, ввёл ли пользователь свои параметры или нет. В листинге 7.12 вновь показана программа `name.py`, изменённая таким образом, чтобы учитывать ввод пользователем аргументов в командной строке.

Листинг 7.12. Измененная программа `name.py`

```
#!c:\python\python.exe
import sys
import string
import leap2
if __name__=="__main__":
    if len(sys.argv) < 2:
        print "Usage:", sys.argv[0], "year year..."
        sys.exit(1)
    else:
        for i in sys.argv[1:]:
            y=string.atoi(i)
            j=leap2.julian_leap(y)
            g=leap2.gregorian_leap(y)
            if j!=0:
                print i, "is leap in the Julian calendar."
```

```
else:
    print i, "is not leap in the Julian calendar."
if g!=0:
    print i, "is leap in the Gregorian calendar."
else:
    print i, "is not leap in the Gregorian calendar."
```

***Прим. В. Шипкова: как-то раз со мной приключилась неприятная штука - перестала передаваться строка параметров. Как оказалось в последствии, это испортила настройки одна из программ-пакетов для Python. Эту беду удалось победить путём простой переустановки Питона.**

Выполнение программы name.py без указания аргументов должно закончиться выводом результата, показанного в листинге 7.13 (на компьютере с UNIX результат может незначительно отличаться) .

Листинг 7.13. Результат выполнения измененной программы name.py

```
C:\Python>python name.py
Usage: name.py year year year...
C:\Python>
```

Программа выводит строку Usage: name.py year year year... (Использование: год год год...), в которой объясняет пользователю, как её следует использовать. Чтобы определить високосный год, пользователь должен ввести его как аргумент в командной строке. Можно также ввести сразу несколько значений, разделяя их пробелами. Результат выполнения программы в случае ввода в строке с аргументом значения 1900 будет выглядеть так, как показано в листинге 7.14.

Листинг 7.14. Результат выполнения программы name.py с введенным аргументом

```
C:\Python>python name.py 1900
1900 is leap in the Julian calendar.
1900 is not leap in the Gregorian calendar.
C:\Python>
```

Теперь Вы имеете программу, которая сообщит Вам, является ли любой интересующий год високосным в обоих календарях (юлианском и григорианском). Имеет смысл теперь переименовать её, например в ly.py, поскольку имя name.py больше не соответствует сути программы. Значение года, вводимое в списке аргументов командной строки, может быть

представлено любым целым числом, естественно, с учётом ограничений, присущих этому типу данных. Так, значения не могут быть больше, чем `sys.maxint`, или меньше, чем `-(sys.maxint-1)`. В следующей главе при рассмотрении использования исключений я покажу Вам, как можно предотвратить подобные ошибки в ходе выполнения программы.

А как определить, был ли високосным какой-либо год до нашей эры? Вопрос о том, как вычислять даты, предшествующие времени ввода календаря в употребление, действительно интересен как в прикладном, так и в философском плане. Поскольку григорианский календарь впервые стал использоваться с 1582 г., а датой введения юлианского календаря считается 707 A.U.C. (*ab urbe condita* – "от основания города"; городом, конечно же, является Рим), можно было бы весьма обоснованно принять решение, что лишено всякого смысла использовать любой из этих календарей для времени, предшествующего соответствующим датам их введения. Тем не менее у астрономов, археологов, богословов, студентов и других заинтересованных лиц часто возникает потребность выстроить хронологию событий, происходящих за долго до появления современного календаря, и сопоставить эти события с нашим временем. Обычно в таких случаях годы нумеруются в обратном порядке. Термин "до Рождества Христова" будет звучать несколько противоречиво, если Вы попытаетесь датировать события, происходившие, к примеру, в жизни Иосифа и Марии. Поэтому многие предпочитают более нейтральный термин "до нашей эры", или сокращенно – до н.э. Спорным остаётся вопрос, какой год считать первым годом нашей эры, тот, в который произошло Рождество Христово, или следующий за ним. Астрономы считают год Рождества Христова нулевым, следующий за ним – первым, а предшествующий – минус первым. В системе, использующей обозначение В.С., год, предшествующий первому году нашей эры (A.D.), является 1-м годом В.С., тогда как в обычной предваряющей (*proleptic*) системе годом, предшествующим 1-му году С.Е., является нулевой год (0), а годом, предшествующим ему, является –1. С позиции математики такой подход имеет глубокий практический смысл. (Кстати, если признать, что существует год 0, то новое тысячелетие действительно наступило 1 января 2000 года, хотя многие с удовольствием отметили приход нового тысячелетия дважды.) Обратите внимание, что программа `ly.py` также прекрасно работает и с отрицательными годами.

Резюме

В этой главе мы рассмотрели функции и модули, а именно: определение собственных функций, создание собственных

модулей, проверку имени модуля использование аргументов командной строки. В следующей главе мы познакомимся со всякими мелочами, которые, тем не менее, приятно скрашивают жизнь программиста.

Практикум

Вопросы и ответы

Почему именно `def`? Почему не `define`?

Как и все остальные программисты, Гуидо не любит набирать на клавиатуре лишние символы.

Имеется ли где-нибудь реальная необходимость в применении инструкции `goto` и функции `long_jump`?

Только не в хорошо продуманном языке. Тем не менее даже в самом С, который многие считают вершиной современного программирования, иногда встречаются инструкции `goto`, которые действительно делают код более ясным и понятным. Можно сказать, что без них он утратил бы свою читабельность. Применения функции `long_jump` требуют некоторые утилиты, написанные также на языке С, особенно утилиты оболочки и интерпретаторы команд.

Прежде я уже слышал об ассемблере. Что это такое? Придется ли мне когда-нибудь изучать его?

Поскольку ассемблер на одну ступеньку выше реального машинного языка, он не использует компилятор. Вместо него он использует транслятор. Это объясняется тем, что буквенные коды, используемые в ассемблере, не требуют вызова большого количества других функций, а просто являются мнемоническими обозначениями, соответствующими конкретным двоичным комбинациям, которые как раз и поддаются трансляции непосредственно компьютером. Знание ассемблера может понадобиться только в том случае, если Вы планируете писать быстродействующие программы, выполняющиеся в режиме реального времени, например высокоскоростные игры, или если Вы захотите писать программы, которые использовались бы в космических проектах. Загляните на Web-страницу по адресу <http://www.s-direkt.net.de/homepages/neumann/sprachen.htm> и найдите там пример программы "Hello, World!", написанной на ассемблере. Вы сразу поймете, что она написана не на Python.

Контрольные вопросы

1. Как в Python определяется функция?

- а) `defun имя(аргументы) (тело)`
 - б) `def имя(аргументы):`
 тело
 - в) `имя(аргументы){тело}`
 - г) Любой из вышеперечисленных способов.
2. Если Юлий Цезарь сделал так, что начиная с 46 года до н.э. каждый человек в Римской империи пользовался его календарем, то какой год соответствует основанию Рима?
- а) 707 год до н.э.
 - б) 753 год нашей эры.
 - в) 753 до н.э.
 - г) 0-й год
3. Как выполняется проверка имени модуля?
- а) В модуле проверяется соответствие значения переменной `__name__` имени модуля, после чего выполняется код модуля.
 - б) В модуле проверяется, не является ли `"__main__"` значением переменной `__name__`, и если является, выполнять код модуля.
 - в) Проверка соответствия переменной `__name__` значению `__main__` выполняется не в модуле, а в основной программе.
 - г) Не стоит об этом беспокоиться.

Ответы

- 1. б. В Python для определения функции используется синтаксис: `def имя(аргумента): тело.`
- 2. в. Годом основания Рима считается 753 год до н.э., или 752-й год по григорианскому календарю.
- 3. б. Если переменная `__name__` имеет значение `"__main__"`, то выполняется код модуля. Проверку имени модуля нужно выполнять обязательно во всех модулях.

Примеры и упражнения

Чтобы получить дополнительную информацию о календарях, посетите Web-страницу по адресу <http://www.calendarzone.com/>. Обширный перечень часто задаваемых вопросов (и ответов на них) по теме летоисчисления в разных календарях можно найти на Web-странице, которая находится по адресу <http://www.pauahtun.org/CalendarFAQ/>; её редактором является Клаус Тондерлинг (Claus Tenderling).

Дополнительную информацию об утилитах оболочки UNIX, в которых используется функция `long_jmp`, можно найти в книге Марка Дж. Рочкинда (Marc J. Rochkind) *Advanced UNIX Programming*, Prentice-Hall. Но для изучения материала этой книги потребуется хорошее знание языка C. Зато потом Вы сможете создавать свои собственные пользовательские оболочки для UNIX.

8-й час

Полезные средства и приёмы программирования

В предыдущих семи главах Вы познакомились практически со всеми основными функциями, операторами и типами данных, имеющимися в Python. В данной главе мы рассмотрим несколько тем, которые трудно отнести к какой-либо из рассмотренных ранее категорий средств программирования. В первую очередь мы обсудим три встроенные функции и одну инструкцию, которые могут оказаться весьма полезными:

- `map()`
- `reduce()`
- `filter()`
- `lambda`

Эти четыре средства чрезвычайно важны для написания полноценных программ на языке Python и лежат в его концептуальной основе.

О концептуальности языка программирования говорят в том случае, если в его выражениях прослеживается чёткая логика и стиль мышления, а не только манипулирование командами. Концептуальность языка побуждает программиста к написанию чёткого, понятного и структурированного кода программы.

В этой главе мы также обсудим следующие темы:

- вывод данных на экран (инструкция `print`);
- форматирование вывода (оператор `'%'`);
- операции ввода-вывода с файлами;
- отслеживание ошибок (инструкции `try` и `except`).

Три важные функции и одна инструкция языка Python

Функции, о которых пойдет речь, являются встроенными. Поэтому для их использования нет необходимости импортировать какой бы то ни было модуль. Мы рассмотрим их в порядке важности, по крайней мере, как это видится мне. Впрочем, когда в последней трети книги мы начнём создавать



графический интерфейс пользователя с помощью средств модуля Tkinter, важность инструкции lambda значительно возрастёт.

Под графическим интерфейсом пользователя понимают строки меню, диалоговые окна и прочие графические объекты, с помощью которых пользователь получает возможность взаимодействовать с приложением. Простейшие программы обходятся без графического интерфейса и управляются из командной строки. К таковым относились все программы под DOS, а окна сеанса DOS, в которых вводятся имена программ и другие команды, являются интерфейсами командной строки.

функция map()

Функция map является, пожалуй, самой лёгкой в применении и простой для понимания. Она вызывает функцию, название которой указывается в аргументе, для всех элементов последовательности, также представленной в строке аргументов. Вот синтаксис функции map() :

`map(функция, последовательность)`

Предположим, у Вас имеется строка символов, полученная из другой программы или записанная в формате, который не поддерживается программой. Например, Вам может потребоваться распаковать строку символов, содержащую значения даты. Первым шагом в решении этой задачи может быть разбиение строки на составляющие элементы, отделенные пробелами. Рассмотрим стандартную строку вывода даты, например ту, которая отображается на экране дисплея UNIX командой date. Эта строка выглядит приблизительно так: Thu Aug 5 05:51:55 MDT 2001

Тогда для разбиения данной строки на составляющие значения можно применить следующий приём:

```
s=string.split("Thu Aug 5 05:51:55 MDT 2001")
```

В этом случае переменная s будет представлять собой список, содержащий элементы ['Thu', 'Aug', '5', '05:51:55', 'MDT', '2001']

После того как Вы получите этот список, необходимо подвергнуть каждый его элемент определённой обработке. Следовательно, необходимо написать функцию, которая займётся анализом каждого элемента и на основании его результатов выполнит необходимые действия. В листинге 8.1 показан упрощенный вариант анализа.

Листинг 8.1. функция look()


```
def look(s):
    if s[0] in string.digits:
        return string.atoi(s)
    else:
        return s
```

В данном случае функция `look()` выполняет довольно бесхитростный анализ. Тем не менее рассмотрим её подробнее. Данная функция анализирует первый символ переданного аргумента `s` и, если этот символ является цифрой, преобразовывает его в целочисленное значение. В противном случае она возвращает аргумент в том же самом виде, в каком он был получен. Это не всегда удобно, в чем можно убедиться при выполнении этой функции (листинг 8.2).

Листинг 8.2. Выполнение функции `look()`

```
import string
def look(s):
    if s[0] in string.digits:
        return string.atoi(s)
    else:
        return s

x = "Thu Aug 5 06:06:27 MDT 2001"
t = string.split(x)
lst = map(look,t)
print lst
```

При выполнении представленной программы (хоть в окне интерпретатора, хоть в отдельном выполняемом файле) на экране появится негодующее сообщение:

```
ValueError: invalid literal for atoi(): 06:06:27
```

(Ошибка значения: `06:06:27` — недопустимый литерал для `atoi()`).

Другими словами, Python жалуется, что ему велели преобразовать строку `"06:06:27"` в целое число, а он не может выполнить этого задания из-за присутствия в ней двоеточий. В листинге 8.3 приведена немного улучшенная версия функции `look()`.

Листинг 8.3. Исправленный вариант функции `look()`

```
def look(s):
    if ':' in s:
```

```
ts = string.split(s,':')
return map(string.atoi,ts)
if s[0] in string.digits:
    return string.atoi(s)
else:
    return s
```

Внесите эти исправления в код листинга 8.2 и вновь выполните программу. На этот раз на экране должна появиться следующая строка: ['Thu','Aug',5,[6,6,27],'MDT',2001]

Обратите внимание, что значение времени ("06:06:27") было преобразовано в список [6, 6, 27], вложенный в список, возвращённый функцией map(). Происходит следующая последовательность действий. Функция map() вызывает функцию look() для каждого элемента списка t, который мы получили с помощью функции string.split(). В свою очередь, функция look() с помощью функции string.atoi() преобразовывает все цифровые элементы списка в целые числа.

Функция map() успешно применяется тогда, когда заранее известна исходная структура данных, как, например, в строковых значениях времени или даты, где отдельные значения разделены символами двоеточия или пробелами. Для дат в календаре Майя вполне подошел бы следующий формат записи: "12.19.6.7.12". Преобразование подобных строк в списки целых чисел — идеальное место приложения функции map:

```
s="12.19.6.7.12"
ls = string.split(s, '.')
md = map(string.atoi, ls)
```

После выполнения этих строк кода переменная md будет содержать список [12,19,6,7,12].

Далее на страницах этой книги я покажу несколько других полезных функций обработки строковых значений времени и даты. А пока перейдем к функции reduce().

функция reduce()

Эта функция предназначена для приведения указанной последовательности значений к одному значению с помощью функции, принимающей два аргумента и возвращающей вещественный результат. Вот её синтаксис: reduce(функция, последовательность)

Например, с помощью функции `reduce()` можно эффективно превращать списки строковых значений в одну строку. Простой пример тому представлен в листинге 8.4.

Листинг 8.4. Использование функции `reduce()`

```
x = ['Thu', 'Aug', '5', '06:06:27', 'MDT', '2001']
def strp(x,y):
    return x+' '+y
r = reduce(strp,x)
print r
```

Выполнение этой программы должно привести к выводу на экран воссоединенной строки даты. В листинге 8.5 показан пример с перемножением последовательности чисел.

Листинг 8.5. Числа и `reduce()`

```
n=range(1,11)
def mult(x,y):
    return x*y
f=reduce(mult,n)
print f
```

На этот раз будет выведен результат вычисления факториала от числа 10 (в математике это выражение имеет вид $10!$). В функцию `reduce` передаются список `[1,2,3,4,5,6,7,8,9,10]`, возвращенный функцией `range()`, и имя функции `mult()`. Функция `mult()` принимает в качестве аргументов первые два элемента списка и возвращает результат из умножения. Затем она снова получает два аргумента, одним из которых является её собственным результатом, а вторым – очередной элемент списка. И так повторяется до конца списка. Вызов функции `range(1,11)` возвращает последовательный список целых чисел, начинающийся с 1 и заканчивающийся 10. Как Вы помните, функция `range()` всегда возвращает список, включающий границу слева и исключающий границу справа.

В подобных ситуациях можно воспользоваться инструкцией `lambda`, чтобы значительно уменьшить количество строк кода, который мы обсудим немного позже.

Функция `filter()`

Как и в предыдущих двух случаях, список аргументов функции `filter` состоит из управляющей функции и последовательности значений:

```
filter(функция, последовательность)
```

В данном случае управляющая функция должна возвращать значение 1 или 0 (числа, традиционно применяемые для представления логических значений true (истинно) и false (ложно)), в зависимости от того, хотите Вы или нет, чтобы некоторый элемент исходной последовательности был включён в новую последовательность, возвращаемую функцией `filter()`. В примере, показанном в листинге 8.6, в качестве управляющей используется функция `gregorian_leap()`, с которой мы уже познакомились в предыдущей главе.

Листинг 8.6. Использование функции `filter()`

```
def gregorian_leap(y):
    if (y%400) == 0:
        return 1
    elif (y%100) == 0:
        return 0
    elif (y%4) == 0:
        return 1
    return 0

n = range(1900,2001)
ly = filter(gregorian_leap, n)
print ly
```

При выполнении этой программы формируется список, элементы которого представляют собой перечень всех високосных годов с 1900 по 2000 включительно.

Инструкция `lambda`

Как уже упоминалось раньше, при определённых обстоятельствах можно уменьшить количество строк кода для всех трёх функций, которые мы только что рассмотрели. Для этого применяется инструкция `lambda`. Эта инструкция отмечает начало определения функции особого типа, называемой анонимной. Под термином *анонимная функция* подразумевается функция без имени. Тем не менее, каким бы парадоксальным это не показалось, эту функцию можно вызывать, но только особым способом, отличающимся от вызова обычных функций. Выше в этой главе для одной из своих демонстрационных программ мы определили небольшую функцию, выполняющую перемножение двух чисел:

```
def mult(x,y):
    return x*y
```

Затем мы передали имя этой функции в списке аргументов функции `reduce`. А вот как можно было выполнить ту же самую работу, но с меньшим числом нажатий клавиш:

```
f = reduce(lambda x,y: x*y,n)
print f
```

Аналогично можно было бы поступить и с другими двумя функциями — `map()` и `filter()`. Синтаксис инструкции `lambda` следующий:

```
lambda список_параметров:
    выражение
```

Обратите внимание, что параметры не заключаются в круглые скобки, что характерно для определения обычных функций. В списке параметры отделены друг от друга запятыми. Необходимо, чтобы после двоеточия в этой же строке следовало выражение функции. Любой символ, не используемый в выражениях, будет означать завершение выражения анонимной функции. Выражения представляют собой любую осмысленную совокупность операндов и операторов: `x+y`, `x*y`, `s[9]` и т.д. Кроме того, анонимную функцию можно снабдить заданным по умолчанию параметром, причём практически так же, как это делается для обычных функций:

```
lambda x, y=13: x*y
```

Если в анонимную функцию, заданную инструкцией `lambda`, не будет передан второй аргумент, то по умолчанию будет использовано значение 13.

Однако, прежде чем использовать инструкцию `lambda`, нужно принять во внимание некоторые предостережения. Первое состоит в том, что эта инструкция затрудняет чтение программы, поэтому её использование должно быть обоснованным. Вторая трудность состоит в том, что анонимные функции не имеют такого широкого доступа к элементам текущего пространства имён, как обычные функции (о пространствах имён подробно рассказывается в главе 11).

***Прим. В. Шипкова: функция `lambda()` довольно экзотична. И выше указаны её недостатки. НО! В тесте перемножения двух чисел, именно из-за усеянного пространства имён она быстрее выделенной функции на 30%. А это существенно! Впрочем, без всяких функций перемножение двух чисел с присвоением длилось на 47% быстрее чем с применением функции, и на 24% быстрее чем при применении**

функции `lambda()`. Также следует учесть: эта функция в качестве тела цикла принимает только ОДНО выражение.

В примере, показанном в листинге 8.7, анонимная функция, заданная инструкцией `lambda`, будет работать не так, как можно было предположить.

Листинг 8.7. Ошибочное использование анонимной функции

```
def myfunc(x):  
    increment = x + 1  
    return lambda increment : increment + 1
```

Дело в том, что инструкция `lambda` ничего не знает о новом приращении переменной. Она знает только об именах переменных, передаваемых в качестве параметров, которые не являются частью определения другой функции. Тем не менее эту проблему можно обойти с помощью аргументов, заданных по умолчанию. С несколькими такими приёмами Вы познакомитесь в завершающей части этой книги.

И наконец, Вы не можете использовать другие инструкции в выражениях анонимной функции, заданной инструкцией `lambda`. Это означает, что из анонимной функции нельзя выполнять печать с помощью инструкции `print`. Это ограничение также можно обойти, о чём Вы узнаете ниже, в разделе "Объекты файлов".

Вывод на экран и форматирование

Вы уже много раз встречались с инструкциями `print`, но мы их использовали, практически ничего не зная о том, как они работают. Вам уже известно, что если в текст программы поместить инструкцию `print`, то она просто отобразит в окне терминала DOS все переменные и константы, указанные за ней:

```
x = ['Thu', 'Aug', '5', '06:06:27', 'MDT', '2001']  
print x
```

В результате выполнения представленного выше кода на экране появится следующая информация:

```
['Thu', 'Aug', '5', '06:06:27', 'MDT', '2001']
```

Как видите, что имели, то и получили. Примерно то же говорил первый президент Украины.

***Прим. В. Шипкова: вообще-то это старый армейский оборот,**

который на публику вынес Черномырдин (ему же нагло и приписывается авторство журналистами.) :-)

Оказывается, инструкции `print` проще всего работать со строковыми переменными, так как это именно тот тип данных, который она выводит. Тогда интерпретатору больше ничего не остаётся делать, кроме как непосредственно передать на дисплей уже готовую строку символов. Числа автоматически преобразовываются в строки символов, так же, как и списки, наборы и словари. Имена функций, однако, выводятся немного иначе. Если есть функция под именем `myfunc` и Вы пытаетесь вывести (обратите внимание, вывести, а не вызвать) её на экран с помощью `print`, на дисплее отобразится сообщение, подобное следующему: `<function myfunc at 7cf8b0>`

Скорее всего, это не то, что Вы предполагали увидеть. Если функция возвращает значение, то можно вывести это значение таким выражением:

```
print myfunc()
```

В этом случае происходит вызов функции `myfunc()`, а затем на экран выводится возвращенное значение.

Как бы то ни было, потребность преобразования в строку символов чисел, последовательностей и словарей будет возникать довольно часто. Несомненно, существует немало способов выполнить эту задачу. Но самый быстрый и простой состоит в том, чтобы заключить всё, что необходимо преобразовать в строку, между символами обратного удара ('') (Данный символ, который в английской терминологии ещё называют *backticks*, вводится той же клавишей, что и символ тильда (~). Эта клавиша для большинства раскладок клавиатуры находится сразу под клавишей <Esc>. Не спутайте символ обратного удара с одинарной кавычкой, это принципиально важно.) Вот пример быстрого преобразования числа в строку:

```
x = 23
y = `x`
```

Переменная `x` содержит число 23, тогда как переменная `y` теперь содержит строку символов "23". Это действительно обычная строковая переменная, которую можно обрабатывать как строку со всеми доступными операторами, например с помощью оператора индексирования. Эту же операцию можно реализовать с помощью встроенной функции `str()`:


```
x = 23
y = str(x)
```

Можно также воспользоваться встроенной функцией `str()`. Отличие между `str()` и `repr()` едва различимо: `str()` возвращает строковое представление переданных ей данных, тогда как `repr()` возвращает строковый объект, который затем можно преобразовать в исходное значение. Для чисел, последовательностей и словарей между этими функциями в принципе нет никакой разницы. Когда мы будем изучать определяемые пользователем объекты, то обсудим более обстоятельно, что ещё полагается выполнять функции `repr()`, хотя и в этом случае отличие невелико.

Иногда возникает необходимость взять несколько переменных различных типов и объединить их в одну строку, возможно, даже вместе с союзами или знаками пунктуации, расставленными между ними. Конечно, это можно сделать с помощью оператора конкатенации строк, т.е. символа `(+)`, как показано в листинге 8.8.

Листинг 8.8. Конкатенация строк

```
x = 20
y = 13
z = "x is '"+str(x)+"", y is '"+str(y)+"" and x * y is '"+str(x * y)
print z
```

Существует более совершенный метод решения подобных задач, который называется *форматированием*. Он не оказывает никакого влияния на оператор `print`, а состоит в использовании специального оператора форматирования (`%`). Код, показанный в листинге 8.9, является примером вывода той же самой строки, которую мы только что получили, но на этот раз с помощью оператора форматирования.

Листинг 8.9. Форматирование с помощью оператора %

```
x = 20
y = 13
print "x is %d, y is %d and x*y is %d" % (x, y, x*y)
```

Символ `*` выполняет функцию оператора форматирования, если слева от него находится строка, а справа — набор. Операнды, находящиеся справа, необходимо записывать в виде набора (т.е. заключать в скобки) только в том случае, если количество элементов превышает единицу. Но значительно спокойнее и безопаснее завести привычку всегда заключать

правый операнд в скобки. Присутствие скобок никогда не будет ошибкой, а отсутствие — может быть. Внутри строки левого операнда также используются символы %, причём их должно быть столько же, сколько элементов находится в наборе правого операнда. Символы % в паре с каким-то другим буквенным символом называются *спецификаторами форматирования*. Например, в предыдущем примере Вы могли заметить, что в строке трижды используются комбинации символов 'Id'. Это спецификаторы формата целых чисел. Каждый спецификатор указывает, что соответствующий ему параметр в наборе, расположенный справа от оператора форматирования, должен быть представлен целым числом. В табл. 8.1 приведены наиболее полезные спецификаторы форматирования.

Таблица 8.1. Спецификаторы форматирования

Символы	Формат
%c	Строка длиной 1 символ
%s	Строка любой длины
%d	Целое десятичное число
%u	Целое беззнаковое десятичное число
%o	Целое восьмеричное число
%x или %X	Целое шестнадцатеричное число (в случае %x — в верхнем регистре)
%e, %E, %f, %g, %G	Числа с плавающей запятой в различных стилях
%%	Символ процента

Спецификатор форматирования допускает также применение специальных инструкций, расположенных между символом процента и управляющим символом. Например, наличие какого-нибудь числа означает команду дополнить пробелами данное поле до указанного числа символов. Полный список спецификаторов форматирования, включающий все комбинации, получится весьма обширным и довольно сложным. В большинстве случаев Вам вряд ли понадобится использовать большую их часть, хотя немного позже я покажу Вам несколько трюков со спецификаторами форматирования. Но пока на этом мы завершим их рассмотрение.

Оператор форматирования можно использовать в любом месте, где Вам заблагорассудится, а не только после инструкции print. Символы I используются всюду, где происходит форматирование строк. Так, оператор форматирования может быть весьма полезным для обработки данных, вводимых пользователем. Польза его применения состоит в том, что в

тех случаях, когда Вы не знаете, к какому типу принадлежат введенные данные, всегда можно воспользоваться спецификатором ' %s', чтобы привести данные любого типа к строке символов. Если переданный аргумент не является строкой, то к нему автоматически будет применена функция `str()`. Более того, в строку символов будут успешно преобразованы даже те данные, которые нельзя преобразовать с помощью функции `str()`. Пример такого преобразования показан в листинге 8.10.

Листинг 8.10. Вывод функции как переменной

```
def myfunc():  
    pass  
  
z = "Printing myfunc yields '%s'." % (myfunc)  
print z
```

Обратите внимание, что мы не вызываем функцию `myfunc()`, а только использовали её имя. Если выполнить в интерпретаторе представленный выше код, то на экран будет выведена строка, подобная этой:

Printing myfunc yields '<function myfunc at 7f74f0>'.

В приведённой выше строке числа (в шестнадцатеричном представлении), следующие после слова "at", сообщают адрес ячейки, начиная с которой в памяти компьютера располагается функция `myfunc()`. Обычно это бесполезная информация, но она может очень пригодиться в некоторых ситуациях во время отладки программы. Вопросы отладки не рассматриваются в этой книге, хотя в последней главе Вы найдёте несколько советов по этому поводу. Дело в том, что Python является настолько ясным и понятным языком, что в большинстве случаев не имеет смысла выносить отладку программы как отдельный этап работы над проектом (а именно так и приходится делать, работая, например, с языком C). Сообщения об ошибках интерпретатор показывает по мере того, как Вы их допускаете, и для обнаружения и исправления ошибок часто бывает достаточно внимательно просмотреть подозрительный участок кода. Хотя во время работы над действительно большими проектами этап отладки будет необходим. С дополнительной информацией по этому вопросу можно познакомиться на домашней Web-странице Python по адресу <http://www.python.org/>.

Ещё один полезный приём, который Вам пригодится в дальнейшей практике, приведён в листинге 8.11.

Листинг 8.11. Форматирование элементов словаря при выводе на печать

```
#!C:\PYTHON\PYTHON.EXE
import math

d={"pi":math.pi,"e":math.e,
  "pipi":2*math.pi,"ee":2*math.e}

print "pi %(pi)s e %(e)s 2pi
      %{pipi}s 2e %(ee)s" % d
```

Оператор форматирования (%) может работать не только с набором, но и со словарем. Обратите внимание, что переменная типа словарь не заключается в скобки. Для форматирования элементов словаря используется следующий синтаксис: %(ключ)спецификатор

Другими словами, Вы указываете функции форматирования, чтобы она произвела поиск элемента в словаре d по заданному ключу и применила к этому элементу указанный спецификатор. Очень часто выражения с оператором форматирования, в которых вместо набора используется словарь, более понятны и читабельны.

Объекты файлов

В большинстве случаев для отображения на экране результатов выполнения своих программ Вы будете полностью удовлетворены возможностями обычной инструкции print. Однако иногда возникает необходимость сохранить полученные данные для дальнейшего использования, ввести исходные данные из файла или открыть файл для изменения. Чтобы выполнять эти операции, Вам необходимы средства управления файлами — объекты файлов.

Совершенно не подозревая об этом, раньше Вы уже использовали так называемые объекты файлов. Это происходило каждый раз, когда применялась инструкция print. При этом использовался объект вывода stdout (ещё одно наследие языка C). Компанию ему составляют ещё два подобных объекта файлов: stdin и stderr. Первый обеспечивает ввод данных по указанной позиции, второй отвечает за вывод сообщений об ошибках. Инструкция print "знает" только, как записывать сообщения в объект stdout. Для использования других объектов необходимо применять иные инструкции. Оказывается, что подобно спискам, объекты файлов обладают методами,

которые можно использовать для реализации этих задач. Все три стандартные объекта файлов, `stdin`, `stdout` и `stderr`, находятся в модуле `sys`. Именно поэтому для их использования необходимо заблаговременно импортировать модуль `sys`.

Стандартные объекты файлов автоматически открываются Python при загрузке модуля. Все остальные объекты файлов необходимо открывать перед их использованием. Для этого используется специальная функция `open()`, что очевидно по её названию. Она является встроенной функцией Python и для её использования не нужно импортировать никакие дополнительные модули. Таким образом, создание и открытие пользовательских объектов файлов является встроенным свойством Python.

Открыть файл очень просто. Оказывается, что значительно сложнее запомнить порядок следования параметров в функции `open()`:

```
x = open("test.txt", "r")
```

Вот и всё, что необходимо сделать, чтобы открыть файл для чтения. Символы, используемые в качестве второго параметра, приведены в табл. 8.2.

Таблица 8.2. Управляющие параметры функции `open()`

Символ	Значение
r, rb	Открывает файл для чтения
w, wb	Открывает файл для замещения исходных данных
a, ab	Открывает файл для добавления к исходным данным

***Прим. В. Шипкова: также есть режим 'r+' – файл открывается для чтения и записи одновременно.**

Дополнительный символ `b` сообщает операционной системе (ОС), что файл нужно открыть в двоичном формате. Это имеет значение только для тех ОС, которые по-разному обрабатывают файлы в текстовом и двоичном форматах. (Разница между ними состоит в том, что в текстовых файлах информация представлена текстовыми символами, а в двоичных – машинными командами.) Системы Windows и Mac различают форматы файлов, тогда как для UNIX это совершенно безразлично. Даже если Вы работаете только под управлением UNIX, имеет смысл завести привычку открывать файлы в двоичном формате. Тем самым Вы обеспечите лучшую переносимость своих программ. Основная проблема, возникающая при переходе от системы к системе, – это различная обработка символов разрывов строк.

Но разрывы строк присутствуют только в текстовом формате, но не в двоичном. Имеет смысл всегда выяснять, в какой именно ОС были созданы переданные Вам файлы. Также при работе с файлами полезно оставлять за собой контроль над тем, какими символами указываются разрывы строк.

Объекты файлов владеют небольшим числом методов и лишь некоторые из них действительно будут Вам полезны на данном этапе. Эти методы вместе с пояснениями и примерами их использования перечислены в табл. 8.3. В данной таблице буквой *f* обозначен объект файла.

Таблица 8.3. Методы объектов файлов

Метод	Пояснение	Пример
<code>f.read()</code>	Считывает весь файл в буфер обмена	<code>buf = f.read()</code>
<code>f.readline()</code>	Считывает в буфер одну строку. В качестве строки подразумевается вся последовательность символов вплоть до символа разрыва строки, характерного для текущей ОС	<code>l = f.readline()</code>
<code>f.read(n)</code>	Считывает <i>n</i> символов в буфер обмена или весь файл, если он содержит меньше строк	<code>c=f.read(1024)</code>
<code>f.write (s)</code>	Записывает в файл строку <i>s</i>	<code>f.write("Hello, World! \n")</code>
<code>f.writelines (list)</code>	Записывает в файл список строк <i>list</i>	<code>list=["Hello, ", "World\n"] f.writelines(list)</code>

Все методы чтения возвращают пустую строку, как только они достигают конца файла. Обратите внимание, что методы записи, в отличие от инструкции `print`, не добавляют автоматически символы разрыва строки или возврата каретки в конце любой записанной строки. Вам придётся делать это самостоятельно. В листинге 8.12 показан код полнофункциональной, хотя и очень простой программы, которая только лишь считывает строки из файла и записывает каждую строку в объект `sys.stdout`.

Листинг 8.12. Программа `readit.py`

```

#!c:\python\python.exe
import sys
if __name__ == "__main__" :
    if len(sys.argv)>1:
        f=open(sys.argv[1],"rb")
        while 1:
            t=f.readline()
            if t=="":
                break
            if t[-1:] == '\n':
                t=t[:-1]
            if t[-1:] == '\r':
                t=t[:-1]
            sys.stdout.write(t + '\n')
        f.close()

```

Эту программу можно использовать в качестве простой утилиты преобразования текстовых файлов. Она считывает входной файл, имя которого указывается в командной строке, и копирует каждую строку в объект stdout. Независимо от того, как завершаются строки исходного файла, каждая строка вывода завершается корректным для текущей операционной системы символом разрыва строки. Обратите внимание, что даже если в файле присутствуют пустые строки, они все равно содержат стандартные для данной ОС символы разрыва строки. И только когда файл будет полностью прочитан, метод readline() возвратит действительно пустую строку ('').

Инструкции try и except.

Прежде чем двигаться дальше, необходимо запустить на выполнение программу readit.py и указать при этом имя несуществующего файла. В результате Вы должны получить на экране сообщение, наподобие приведенного в листинге 8.13.

Листинг 8.13. Выполнение readit.py

```

C:\Python>python readit.py shootka.txt
Traceback (innermost last):
File "readit.py", line 5, in ?
  f=open(sys.argv[1],"rb")
IOError: [Errno 2] No such file or directory: 'shootka.txt'

C:\Python>

```

В тех случаях, когда Вы пишете программу не для себя, а для других пользователей, то, чтобы не пугать своего будущего пользователя невразумительными сообщениями на

экране, имеет смысл позаботиться о том, чтобы снабдить свою программу более информативными сообщениями о возможных ошибках. Именно решением этой задачи мы сейчас и займёмся.

Синтаксис инструкции `try` довольно прост:

```
try:  
    выражения, которые нужно опробовать...
```

Обратите внимание, что выражения блока инструкции `try` должны иметь отступ. Инструкция `try` не работает сама по себе и обязательно требует присутствия ещё одной инструкции — `except`. Причем `except` должна иметь тот же отступ, что и связанная с ней `try`. Если Вы уже запутались в отступах, посмотрите на листинг 8.14, где показана измененная программа `readit.py`.

Листинг 8.14. Измененная программа `readit.py`

```
#!/c:/python/python.exe  
import sys  
if __name__ == "__main__" :  
    if len(sys.argv)>1:  
        try:  
            f=open(sys.argv[1],"rb")  
        except:  
            print "No file named %s!" % (sys.argv[1],)  
        while 1:  
            t=f.readline()  
            if t=="":  
                break  
            if t[-1:] == '\n':  
                t=t[:-1]  
            if t[-1:] == '\r':  
                t=t[:-1]  
            sys.stdout.write(t + '\n')  
        f.close()
```

К сожалению, выполнение этой программы не пройдет гладко. Жалоба интерпретатора показана в листинге 8.15.

```
C:\Python>python readit.py shootka.txt  
No file named shootka.txt!  
Traceback (innermost last):  
  File "readit.py", line 10, in ?  
    t=f.readline()  
NameError: f
```


C:\Python>

Что же опять произошло? Сначала, как и предполагалось, было выведено наше сообщение об ошибке, но затем Python сообщил ещё об одной ошибке, которую мы никак не ожидали.

Внимательно просмотрев текст программы, Вы заметите, что в случае сбоя при выполнении блока оператора `try`, в результате чего и вызывается *исключительная ситуация* (технический термин, используемый при возникновении ошибки в Python), переменная `f` никогда не будет создана. Все было бы прекрасно, если бы мы не забыли, что вслед за выводом сообщения об ошибке надо прекратить выполнение программы. В нашем же варианте программа после возникновения ошибки продолжает выполняться дальше, что ведёт к следующей ошибке. Когда происходит вызов метода `f.readline()`, вдруг обнаруживается, что объекта `f` нет и в помине. Простое изменение, показанное в листинге 8.16, решит эту проблему.

Листинг 8.16. ещё раз измененная readit.py

```
#!/c:\python\python.exe
import sys
if __name__ == "__main__" :
    if len(sys.argv)>1:
        try:
            f=open(sys.argv[1],"rb")
        except:
            print "No file named %s!" % (sys.argv[1],)
            sys.exit(0)
        while 1:
            t=f.readline()
            if t=='':
                break
            if t[-1:] == '\n':
                t=t[:-1]
            if t[-1:] == '\r':
                t=t[:-1]
            sys.stdout.write(t + '\n')
        f.close()
```

Вызов функции `sys.exit(n)` является общепринятым методом аварийного завершения программ. Если с аргументом `n` передаётся 0, это означает, что программа завершилась нормально. Любое другое значение сообщает операционной системе, что произошел какой-то сбой. Программисты редко используют вызов `sys.exit(0)`. Если Python до конца

прочитывает весь файл программы и больше не встречается ни одной команды, он предполагает, что программа была выполнена корректно, поэтому сам осуществляет вызов `sys.exit(0)`.

В нашем случае, если пользователь укажет в параметрах программы несуществующее или неправильное имя файла, Вы можете трактовать это либо просто как обычную опечатку, либо как ошибку. Я, например, предпочитаю трактовать подобную ситуацию как опечатку, поэтому вызываю `sys.exit(0)` вместо `sys.exit(-1)`.

Исключительные ситуации (или проще — *исключения*) — это тот мост, который связывает программный продукт в идеале с его реальным воплощением и эксплуатацией пользователями. Для управления исключениями в Python имеются встроенные инструкции `try` и `except`, которые не только дают возможность программистам отслеживать ошибки, но и позволяют упростить код программы и сделать его более логичным.

Вы помните, как в листинге 8.2 у нас возникла проблема из-за того, что `string.atoi()` была передана строка, символы которой не были цифрами. Когда функция `atoi()` не получила ожидаемые значения, Python прервал выполнение программы и сообщил об ошибке. Одним из методов корректного завершения работы программы состоит в том, чтобы первым исследовать строку на наличие в ней недопустимых символов. Если таковые будут обнаружены, то следует вывести для пользователя информативное сообщение об ошибке. Очень здорово, но если функция `string.atoi()` уже сама знает, какие символы ей подходят, а какие нет, то зачем усложнять себе жизнь и создавать конструкции `try...except`? Ответ простой: хотя бы из уважения к вашим пользователям, которые оценят это. Кроме того, Вы увидите, что код программы станет чётче, понятнее и удобнее для отладки, так как Вы явно укажете в программе места, где могут произойти сбои в работе. И вдобавок — это стиль настоящего профессионального программиста. Если бы профессионал программирования на языке Python писал подобный код, то он выглядел бы так, как показано в листинге 8.17.

Листинг 8.17. Профессионал использовал бы функцию `atoi()` только так

```
s = raw_input("Enter a number: ")
try:
    n = string.atoi(s)
except ValueError:
    print "I require numbers, please."
```



```
sys.exit(1)
...продолжение программа...
```

функция `raw_input()` весьма удобна для считывания данных, вводимых пользователем с клавиатуры. Строковый аргумент используется в качестве приглашения-подсказки, с помощью которого на экран можно вывести наводящие инструкции, если в них появится необходимость.

Синтаксис инструкции `except` более сложный, чем инструкции `try` (наверное, потому, что проще не бывает). С помощью этой инструкции можно не только определять тип ошибки, но и использовать любой собственный код для обработки этих ошибок. Например, можно написать код обработки исключительных ситуаций (это код, следующий за предложением `except` выражение:), который бы отслеживал ошибки при открытии файлов и в зависимости от типа обнаруженной ошибки либо выводил предупреждающее сообщение и продолжал выполнение программы, либо завершал программу. В конструкции `try...except` можно использовать необязательную инструкцию `else`, тело которой будет выполняться в том случае, если в блоке `try` не была обнаружена ни одна ошибка, что также иногда бывает полезно. В тех случаях, когда программист точно знает, какие ошибки могут произойти, вполне подходит упрощенный код обработки исключительных ситуаций, показанный в предыдущем примере.

С инструкцией `try` возможна ещё одна конструкция: `try...finally`. Эта конструкция похожа на предыдущую, только тело инструкции `finally` выполняется в любом случае, независимо от возникновения исключительной ситуации. Хорошим примером использования подобной конструкции является процедура удаления временных файлов. В этом случае код программы помещается после инструкции `try`, а временный файл удаляется в теле инструкции `finally`. В результате, даже если в работе программы произойдет сбой, она перед завершением удалит свои временные файлы, чтобы не засорять ими жесткий диск. Сбой вызовет исключительную ситуацию, которая временно сохранится в памяти компьютера, а на выполнение будет запущен блок `finally`, после чего вновь произойдет вызов исключения и завершение программы. Простой пример такого решения показан в листинге 8.18.

Листинг 8.18. Удаление временных файлов

```
#!c:\python\python.exe
import os
import tempfile
```

```
tf = tempfile.mktemp()
try:
    f = open(tf, "wb")
    f.write("I'm a tempfile.\n")
    f.flush()
    f.close()
finally:
    try:
        os.remove(tf)
    except:
        pass
```

В данном примере функция `mktemp()`, взятая из модуля `tempfile`, возвращает строку символов, которую можно использовать в качестве имени временного файла. Функция `mktemp()` гарантирует уникальность данного имени в текущем каталоге. Как бы ни был назван временный файл, его обязательно нужно удалить в конце сеанса работы программы. Если же программа не сможет открыть временный файл, нужно сообщить об этом пользователю. Конструкция `try` с `except` внутри блока `finally` как раз и предназначена для обработки подобных ситуаций. Инструкция `finally` сообщит пользователю об ошибке. Блок `except...pass` внутри блока `finally` предназначен для того, чтобы предупредить вывод интерпретатором на экран сообщений о всех других возможных ошибках, например о том, что указанный файл не существует. Слишком много информации может только запутать пользователя. Поэтому программа ограничится одним сообщением, удалит временный файл, если он все же был создан, и тихо уйдет со сцены.

Резюме

В данной главе мы рассмотрели несколько концептуальных конструкций Python с функциями `map()`, `reduced()`, `filter()` и инструкцией `lambda`. Вы узнали о способах вывода и форматирования данных на экран, а также об объектах файлов, которые, как оказалось, активно использовались нами и раньше, но были настолько скромными, что не давали о себе знать. Наконец, мы узнали о способах обработки исключений с помощью конструкций `try...except...else` и `try...finally`.

Это последняя глава части I. В следующих семи главах мы подробно изучим объекты и начнём применять на практике кое-что из того, с чем уже успели познакомиться.

Практикум

Вопросы и ответы

Почему lambda? Откуда появилось это название?

Гуидо позаимствовал инструкцию lambda из Scheme — диалекта языка Lisp, хорошо структурированного и функционального языка программирования. Термин lambda как абстрактное представление функций, видимо, впервые использовался в статье Алонзо Черча "Формулирование простой теории типов" (Alonzo Church, A Formulation of the Simple Theory of Types", 1940, *The Journal of Symbolic Logic*). Самое смешное, что этот термин появился в результате типографской ошибки. Первоначально автор статьи для обозначения абстрактной функции использовал символ x со значком \wedge сверху. Но значок сместился влево от x и по ошибке, был принят за греческую букву лямбда (λ). За эту информацию спасибо Чарльзу Г. Валдману (Charles G. Waldman), Майклу Махони (Michael Mahoney), Рене Грогнарду (Rene Grogard) и Максу Хейлперину (Max Nailperin).

Какие ещё средства программирования и логические конструкции лежат в основе концептуальности языка Python?

Следует обратить внимание ещё на такие функции, как `apply()` и `exec()`. Но их изучение выходит за рамки этой книги. Это тема для хорошо подготовленных программистов, а не для новичков. Вам я пока рекомендую хорошо освоить конструкции, описанные в этой главе.

Обработка исключений, как видно, действительно является довольно эффективным подходом к программированию. Используют ли другие языки нечто подобное?

Безусловно. В той или иной форме обработка исключительных ситуаций поддерживается всеми современными языками программирования. Наиболее широко и эффективно исключения используются в языке C++. В то же время нужно заметить, что это в общем-то недавнее приобретение среди широко распространенных средств программирования.

Контрольные вопросы

1. Какие три важных средства программирования лежат в основе концептуальности Python?
 - a) `map()`, `reduce()`, `filter()`
 - б) `compiled`, `exec()`, `apply()`
 - в) `map()`, `reduce()`, `lambda`

- г) `lambda, apply(), exec()`
2. Если одной инструкцией `except` Вам нужно перехватить для обработки несколько исключительных ситуаций, какой, по-вашему, синтаксис для этого подойдёт?
- а) `except Error1 + Error2 + Error3 ...`
- б) `except Error1 and Error2 and Error3 ...`
- в) `except Error1 or Error2 or Error3 ...`
- г) `except Error1 | Error2 | Error3 ...`
3. Каким методом проще всего преобразовать целые числа в строку символов?
- а) `s = str(123)`
- б) `s = '123'`
- в) `s = "%s" % (123)`
- г) `s = string.string(123)`

Ответы

1. а, б, в и г. Это был вопрос с подвохом. Все перечисленные средства важны для поддержания концептуальности программирования, так что любой ответ будет правильным.
2. в. Чтобы перехватить несколько разных исключений, следует использовать выражение `except Error1 or Error2 or Error3....` Исключения, не представленные в этом перечне с оператором `or`, проследуют незамеченными.
3. а, б или в. Единственный неправильный ответ — `string.string()`. Такой функции просто не существует, но все остальные средства вполне подойдут для этой цели.

Примеры и задания

Чтобы узнать больше о спецификаторах форматирования, обратитесь к превосходному справочнику по языку C авторов Харбисона и Стала (Harbison and Steele, *C: A Reference Manual*). Выдержав в настоящее время четвертое издание, эта книга стала настольной для многих программистов. Хотя речь в ней идёт совсем не о Python, Вам она вполне подойдёт, поскольку идея спецификаторов форматирования была заимствована в Python из C. В этой книге Вы найдёте множество примеров написания выражений, демонстрирующих все

возможные сочетания управляющих символов и сопутствующих им атрибутов.

Попробуйте заново переписать функцию `look()`, упоминавшуюся в первом разделе этой главы, чтобы включить в неё конструкцию `try...except` для достижения большей надёжности и логичности кода.

Полный список всех встроенных исключений Python можно найти на Web-странице по адресу <http://www.python.org/doc/current/lib/module-exceptions.html>.

Часть II

Объекты, отраженные в зеркале, ближе, чем кажутся

Темы занятий

9. Объекты как они есть
10. Определение объектов
11. Концепции объектно-ориентированного программирования
12. ещё о концепциях объектно-ориентированного программирования
13. Специальные методы классов в Python
14. Лаборатория доктора Франкенштейна
15. Лаборатория доктора Франкенштейна (продолжение)
16. Объекты в деле

9-й час

Объекты как они есть

С этой главы мы начинаем захватывающее путешествие в мир объектов. Этой теме отведена почётная центральная часть книги. Мы начнём с подготовки и возведения фундамента для восприятия информации, изложенной в последующих главах, но к концу данной главы Вы будете в состоянии определять объекты; представлять переменные как ссылки на объекты; понимать основы объектно-ориентированного программирования.

Изучая материал этой главы, Вы мало времени проведёте за компьютером. Вместо этого мы познакомимся с объектами, а подробности Вы оставим до следующих глав.

Объекты - это всё вокруг

В Python всё является объектами. Объекты можно представить себе в виде особых блоков или ящичков, в которых может храниться всякая всячина. Содержимое встроенных в Python объектов, каковыми являются базовые типы данных, довольно ограничено. Числовые типы, например, не содержат никаких методов, только значение. Списки являются объектами и содержат как методы, так и значения, но нет никакой возможности добавить к ним другие методы. Наборы вообще не имеют методов, но могут содержать значения любого типа. Словари имеют методы и содержат значения, но опять же, отсутствует возможность добавить к ним хотя бы несколько своих методов. Поэтому, хотя в Python всё является объектами, между объектами базовых типов данных и объектами, определяемыми пользователем, есть много отличий. Наиболее яркое отличие состоит в том, что определяемые пользователем объекты могут содержать любые понравившиеся Вам члены. Под членами объекта подразумевается всё, что угодно. Пользовательские объекты, которые ещё называют классами, могут содержать неограниченное количество необходимых Вам данных базовых типов. Кроме того, они могут содержать другие объекты. Наконец, они могут содержать функции. Функции, определённые внутри объектов, называются методами. Модуль также является особым видом объекта, так как может вмещать переменные базовых типов, другие объекты и методы.

В сущности, когда Вы пишете программу на языке Python, то создаёте модуль. Модуль можно импортировать в свою программу с помощью инструкции `import`, указав имя файла, в котором он хранится. Все это Вам уже знакомо. Так, мы использовали модуль `string`, который находится в файле `string.py`. Все члены этого модуля определены внутри этого файла, т.е. имеют свои имена и к ним можно обратиться с помощью идентификатора `string.имя члена`. Таким образом, раньше Вам уже много раз приходилось оперировать объектами и их методами, например, когда Вы использовали функцию `string atoi()`. Модуль, в котором выполняется ваша программа, всегда носит имя `"__main__"`. Как Вы помните, проверяя текущее имя модуля, мы можем выяснить, является ли он в данный момент текущим или импортированным в основную программу:

```
if __name__ == "__main__":  
    выполнение программы модуля или самотестирование  
else:  
    определение членов модуля
```

В Python даже числа являются объектами. Все объекты должны нести с собой некие атрибуты, чтобы Python мог распознать, к какому типу относится тот или иной объект.

Так, все числа имеют идентификатор, который сообщает, например: "Я — комплексное число" (или какой-либо другой числовой тип). Всякий раз, когда Вы вводите какое-нибудь число в код программы, а затем выполняете её с помощью интерпретатора, Python распознает тип числа и ассоциирует (проще — устанавливает связь) атрибут типа с фактическим значением. Но, как Вам уже известно, числа так же не имеют методов, как строки и наборы. Только списки и словари (называемые ещё ассоциативными типами) имеют методы, например `list.append()` или `dict.keys()`. Поэтому ещё раз усвоим, в Python объект объекту рознь. Этот факт не очень нравится Guido со товарищами, и они планируют в будущих версиях Python нивелировать эти различия, обеспечив методами строки и наборы. Станет возможным даже такое выражение, как `'23'.atoi()`. Но это оставим для будущего. Тем более, зная стиль работы Guido, я уверен, что все программы, которые мы создадим на языке Python сегодня, будут также хорошо выполняться в будущих версиях, несмотря на все нововведения. Даже если строки будут снабжены методами, модуль `string` всё равно можно будет использовать параллельно.

То, что работа Python полностью основана на использовании объектов, позволяет ему быть языком с динамическим присвоением типа данных. Если бы значение переменной и атрибут типа не были связаны вместе в один объект, то определение типа переменной (а тем более изменение типа) было бы длительной процедурой. Таким образом, если бы числа в Python не были объектами, то это был бы ещё один язык со статическим распределением типов, каковых уже предостаточно. А медленных языков программирования также предостаточно.

Объекты представляют собой совокупность членов. О любой подобной совокупности, содержащей переменные-члены, говорят, что она обладает *структурой*. Переменные-члены также определяют *состояние* объекта. Чтобы понять смысл этого термина, представим себе обычный выключатель света. Он может быть либо во включенном *состоянии*, либо в выключенном.

***Прим. В. Шипкова: на самом деле я установил, что ещё есть третье состояние — "сломан".**

Большинство переменных характеризуется значительно большим количеством состояний, а не двумя, как выключатель, но это ничего не меняет, а только делает их более удобными в применении.

О любом объекте, который содержит методы, говорят, что он функционален или характеризуется особым поведением. Второй термин звучит не официально, но зато лучше позволяет понять смысл использования объектов. Давайте сравним объект с котом, милым Васькой, который, быть может, сейчас трётся у ваших ног. Вы его погладили, и Васька замурлыкал. Стимулирующее воздействие вызывает ответную реакцию, что и является поведением. Поведение бывает разным и во многом зависит от Вашего обращения. Можно, например, дернуть кота за хвост. Так что если объект разрушит Вашу программу, то виноваты в этом будете Вы сами — не надо было дергать за хвост. Скажите ещё спасибо, что он Вас не поцарапал.

У объектов, конечно, хвоста нет, но предостаточно альтернативных методов стимулировать их работу. В соответствии со сложившейся терминологией объектно-ориентированного программирования такая стимуляция называется *направлением сообщения* объекту. Это абстрактный термин, который на практике в Python сводится к простому вызову методов объекта.

В Python некоторые объекты имеют состояние, но не имеют функциональности. Все числовые типы данных характеризуются только состоянием. Строки символов и наборы также обладают состоянием и, поскольку у них отсутствуют какие бы то ни было методы, являются объектами без поведения. Списки и словари характеризуются и состоянием, и функциональностью: `list`, `append(item)` и `dict.keys()` являются примерами методов, определяющих функциональность данных объектов. Наконец, модули, которые, как правило, не имеют никаких переменных-членов, являются примерами объектов с функциональностью, но без состояния. Кот с состоянием, но без поведения, был бы мертвым котом, но для объектов это допускается. Поэтому аналогию между объектом и котом можно проводить лишь до некоторой степени. В заключение можно лишь сказать, что наиболее интересными и яркими (как живые!) объектами являются те, которые характеризуются как состоянием, так и поведением.

Мы возвратимся к свойствам объектов немного позже, после того как рассмотрим уже знакомые Вам переменные с другой стороны.

Переменные как ссылки на объекты

Во многих языках программирования переменные представляют собой имена, ассоциированные с определёнными ячейками памяти компьютера, в которых хранятся значения переменных.

В Python всё реализовано иначе. Поскольку в Python переменные являются объектами, имя переменной ссылается на объект, а не на физическую ячейку памяти. Python берет на себя всю ответственность за управление распределением памяти в соответствии с его внутренними алгоритмами. Это очень важный момент, поскольку допускается иметь множество ссылок на один и тот же объект. Если объект изменится, то каждая переменная, которая ссылается на него, также изменится соответствующим образом. Не забывайте, что даже целые числа являются объектами. Python заранее резервирует определенный объем памяти, чтобы уменьшить время, необходимое для создания нового объекта. Каждый раз, когда создается новая целочисленная переменная, что в Python всегда сопровождается инициализацией, т.е. присвоением значения, то для первой сотни целых чисел Python ассоциирует новую переменную с соответствующим ей уже существующим объектом. Любая переменная, которой присваивается, например, значение 42, ссылается на один и тот же целочисленный объект. Создание нового целочисленного объекта происходит только в случае присвоения значения, выходящего за сотню, например 1 000 000. Никаких проблем с тем, что разные переменные ссылаются на один целочисленный объект, как правило, не возникает, поскольку невозможно изменить объект 1 на что-нибудь другое.

Но другие объекты не отличаются таким постоянством. Помните, в предыдущих главах мы уже обсуждали отличия между изменяемыми и неизменяемыми (константными) объектами? Числа, строки символов и наборы являются константами, тогда как списки, словари и определяемые пользователем объекты можно изменять. При изменении целочисленной переменной в действительности изменяется её ссылка на объект, но не сам объект. Комплексные числа, хотя и имеют два компонента, каждый из которых может быть считан в отдельности, так же ссылаются на неизменяемые объекты, как и любая другая числовая переменная. Попробуйте выполнить с помощью своего интерпретатора следующие строки кода:

```
>>> i = 3 + 4j
>>> i.real 3.0
>>> i.imag 4.0
>>> i.real = 2.0 Traceback (innermost last):
File "<pyshell#4>", line 1, in ?
i.real = 2.0
TypeError: object has read-only attributes
>>>
```

Вы получили сообщение об ошибке, поскольку попытались изменить константный объект, на который ссылается один из

компонентов комплексного числа. Благодаря константности объектов, на которые ссылаются все числовые переменные, с их использованием в Python редко возникают какие-либо проблемы. Но этого нельзя сказать о списках, словарях и пользовательских объектах, непостоянство которых может приводить к неожиданным результатам. Рассмотрим один из примеров.

В главе 7 Вы узнали, что функции могут иметь заданные по умолчанию значения параметров, которые вступают в силу в тех случаях, когда в вызове функции не представлены соответствующие аргументы. В качестве параметра функция может принимать список, и если никакой список не будет передан ей с аргументом, то, возможно, необходимо будет предусмотреть создание в функции пустого списка. В таком случае можно попытаться использовать следующее определение функции: `def myfunc(l=[]):`

Это не самая лучшая идея, в чем мы убедимся, внимательно проанализировав выполнение следующей коротенькой программы, показанной в листинге 9.1.

Листинг 9.1. Программа `spam.py`

```
#!/ C : \PYTHON\PYTHON . EXE
def spam(n, l=[]):
    l.append(n)

x = spam(42)
return l
print x
y = spam(39)
print y
z = spam(9999, y)
print x, y, z
```

После выполнения этой программы Вы увидите, что все три переменные, `x`, `y`, и `z`, ссылаются на один и тот же объект — список `l`, который первоначально был пустым. Вероятно, Вы не ожидали такого результата, поскольку для каждой из переменных вызывали функцию `spam()` с разными аргументами. Предполагалось, что каждый раз будет создаваться новый список. Чтобы так и было, измените определение функции `spam()` так, как показано в листинге 9.2.

Листинг 9.2. Измененное определение функции `spam()`

```
def spam(n, il=[]):
```

```
flist = il[:]
flist.append(n)
return flist
```

Теперь программа будет работать правильно и возвратит три разных списка. Примененный нами оператор извлечения (`[:]`) создаёт копию пустого списка в случае, если список не будет передан с аргументом в вызове функции. Копия объекта по определению не является тем же самым объектом, что и оригинал. Вы можете непосредственно убедиться в этом с помощью оператора `is`, который возвращает значение 0, если два объекта не совпадают друг с другом, и 1, если операнды ссылаются на один и тот же объект. При использовании нашей первой версии функции `sram()` выражение `x is y` возвратит 1, а после изменения функции – 0. Проверьте это либо с помощью интерпретатора, либо редактора IDLE. Или можно просто изменить программу `sram.py` таким образом, чтобы она выводила на экран результат выражения `x is y`.

С другой стороны, если в программе или модуле вне функции создаётся пустой список, а затем другая переменная, которая также содержит пустой список, как показано в следующем примере:

```
list1 = []
list2 = []
```

то в этом случае два списка будут ссылаться на разные объекты. Так происходит из-за того, что операторы присвоения в коде программы выполняются в Python по мере их обнаружения. Всякий раз, когда встречается определение функции, Python преобразовывает все инструкции функции в специальную форму, в так называемые байт-коды, которые хранятся в памяти для дальнейшего использования. Заданные по умолчанию параметры, перечисленные в списках параметров, обрабатываются таким же образом, как и обычные операторы присвоения, но выполнение этих операторов происходит только однажды при обнаружении определения функции, но не при её вызове. Таким образом, любой объект, требуемый параметром по умолчанию, создаётся только однажды, а при всех последующих вызовах функции происходит обращение только к этому объекту.

В большинстве случаев особенность трактовки переменных в Python как объектов останется для Вас прозрачной и не повлияет на стиль программирования. Гораздо чаще ошибки возникают из-за неправильного применения переменных. Например, если Вы создадите переменную `i`, присвоив ей

целочисленное значение, затем в программе присвоите ей строковое значение, а потом попытаетесь использовать переменную `i` в контексте, требующем только целое число, то возникнет ошибка. Но объектность переменных Python в данном случае не имеет к этому никакого отношения.

Объектно-ориентированное программирование

Объектно-ориентированное программирование, или ООП, является стилем программирования с применением объектов. Мы уже видели, что объекты характеризуются состоянием и владеют функциональностью, но прежде чем приступить к применению их на практике, следует рассмотреть ещё одно свойство объектов — *тождественность*. Под этим термином подразумевается возможность отличить отдельный экземпляр объекта от всех остальных. Возьмём, к примеру, объекты чисел. Одна единица совершенно ничем не отличается от любой другой единицы. Именно по этой причине во всех местах, где в вашей программе необходимо значение 1, Python использует один и тот же объект, который создаётся при запуске интерпретатора. Но единица легко отличается от двойки, поскольку единица не является двойкой. Таким образом, базовую формулировку тождества можно выразить следующим образом: 1 — это 1, но 1 — это не 2.

Хотя всегда можно отличить два несовпадающих объекта, многие из них могут характеризоваться одинаковым состоянием, или одинаковой функциональностью, или и тем и другим. Объекты, которые имеют общую функциональность и отличаются только состоянием, можно создавать с помощью единого шаблона, называемого классом. Классы можно представить себе в виде форм, используемых хозяйками для выпечки тортов или печенья. Представьте, что Вы раскатали тесто и собрались вырезать из него формочкой заготовки для печенья. При этом можно сказать, что Вы создаёте экземпляры печенья, каждый из которых представляет собой реальный объект. Классы сами по себе не являются реальными объектами, а только их образцами. Объекты, созданные с помощью класса, называются экземплярами класса, а процесс создания объекта называется реализацией. Из этого следует, что когда Вы даёте задание Python присвоить переменной какое-нибудь значение, что выполняется с помощью выражения типа `a=42`, то под этим подразумевается реализация объекта 42 в переменную под именем `a`. В данном примере переменную `a` можно представить себе в виде печенья, посыпанного 42 стружками шоколада, или в виде омлета из 42 яиц (кому что больше нравится). Печенье не имеет поведения, кроме того что оно съедается. Напротив, наш первый условный объект — кот — отличается довольно сложным поведением, благодаря

чему может приносить радость или огорчать нас. Работать с объектами в компьютерных программах гораздо проще, чем убедить своего кота не лазить на стол.

Давайте вернёмся к нашим формочкам для печенья. Каждый раз, когда Вы хотите изменить форму печенья, приходится начинать с изготовления новой формы. Вы облегчите свой труд, если удастся видоизменить уже имеющуюся у Вас форму (где-то вытянуть её, где-то подогнуть, закруглить), и можно приступать к выпечке нового печенья. Если изменить форму не удаётся, то приходится с нуля изготавливать новую, что сложнее, но тоже возможно. То же самое справедливо и для объектов. Вы можете определить класс, который характеризуется поведением и состоянием, и использовать его для создания необходимого количества экземпляров объектов. Если в дальнейшем Вы решите, что нужны другие объекты, в основном подобные предыдущим, но имеющие незначительные отличия в состоянии или функциональности, то для повышения эффективности своего труда можно начать с первоначального класса и изменить его структуру, добавив или удалив некоторые члены. Когда один класс получает часть своих членов, определяющих его функциональность и состояние, от другого класса, говорят, что новый класс наследует исходный. Наследование является ещё одним важным свойством объектов, которое обеспечивает возможность повторного использования ранее определённых классов, независимо от того, были ли они написаны лично Вами или другим программистом. Так, импортное использование модулей в программы позволяет использовать функциональность, разработанную другим программистом и сохранённую в импортированном модуле. Хотя это действенная форма повторного использования программных блоков, избавляющая Вас от необходимости повторно переписывать ранее созданные коды функций, с формальной точки зрения этот процесс ещё не является наследованием. Чтобы разобраться в наследовании объектов, этому вопросу следует уделить больше времени. Поэтому давайте обсудим этот вопрос в следующих главах. Для общей информации сообщу Вам только, что в некоторых языках программирования наследование является единственным методом создания классов. Одним из таких языков является Java. В Java каждый определяемый пользователем класс должен наследоваться от некоторого другого класса. В Python, как и в C++, классы можно создавать с нуля, не наследуя их ни от каких других классов.

Прежде чем мы завершим данную главу, имеет смысл рассмотреть ещё два важных свойства объектов — *инкапсуляцию* и *полиморфизм*. Названия путающие, но понять их не составит особого труда.

Инкапсуляция является просто процессом определения интерфейса объекта и способа его выполнения. Раньше Вы уже использовали интерфейсы объектов, например функций. Интерфейсом функции являются способ её вызова и ожидаемый результат. Например, функция `atoi()` модуля `string` требует, чтобы при вызове ей передавали два аргумента: обязательно одну строку и при необходимости один целочисленный аргумент. Если поменять местами эти два параметра в вызове функции `atoi()`, интерпретатор выведет сообщение об ошибке. Таким образом, интерфейс функции `atoi()` определен этими двумя параметрами и возвращаемым значением, которое всегда является целым числом. Объекты также имеют интерфейсы, хотя часто они значительно сложнее, чем у функций.

Вторая часть инкапсуляции состоит в выполнении объекта. Когда функция `string.atoi()` вызывается в полном соответствии с её интерфейсом, то всё, что происходит между вызовом и возвращением значения, называется выполнением функции (или объекта). Другими словами, это весь тот набор инструкций, с помощью которых достигается требуемый результат. Как пользователя Вас совершенно не должно волновать, как именно функция `string.atoi()` выполняет свою работу. Вас интересует только то, как правильно запустить её и какой результат она возвратит.

Однако не так все просто с инкапсуляцией. Любая функция, как правило, имеет только один интерфейс, видимый для пользователя. Определяемые пользователем объекты ведут себя несколько сложнее. Они содержат переменные-члены, определяющие их состояние, и методы, лежащие в основе их функциональности. Большинство языков программирования, поддерживающих классы, обладают средствами управления доступом (видимостью) к различным членам класса для пользователей. Это позволяет программистам скрывать некоторые члены класса от несанкционированного, как правило, ошибочного вмешательства пользователей, тогда как другие члены служат открытым интерфейсом класса. В таких языках, к которым Python не относится, инкапсуляция также включает открытие и закрытие доступа к членам класса. Обычно интерфейс класса делается открытым, а его выполнение — закрытым. Для новичков часто возникает проблема, что отнести к интерфейсу, а что к выполнению. Как показывает мой личный опыт, очень часто в классах, даже тех, которые были созданы профессиональными программистами, оказываются закрытыми именно те члены класса, которые как раз тебе сейчас и нужны.

***Прим. В. Шипкова: на самом деле, защищённые члены класса**

могут пригодиться только в том случае, если Вы собираетесь заработать на программировании, т. е. пожить за счёт своих ближних. ;) [целевые заказы организаций я сюда не отношу, хотя всё ещё остаётся вопрос, сколько программист попросил]. На самом деле, на сопровождении программных продуктов можно заработать раза в 3 больше (не верите? почитайте про проект Зопи.)

В Python ко всем членам класса, будь то переменные или методы, можно обращаться извне без всяких ограничений, лишь бы эти члены действительно присутствовали в классе. В Python не существует ни единого способа предотвратить "несанкционированное проникновение" в класс, чем он сильно отличается от C++, где программист может запретить даже просмотр имеющихся членов класса, не то что их использование и изменение. В языке Python используется совершенно иной подход к инкапсуляции, чем в других языках объектно-ориентированного программирования. В Python, чтобы открыть интерфейс, нужно сообщить пользователям, какие методы и переменные они могут вызывать или изменять, а чтобы закрыть некоторые члены, опять-таки необходимо сообщить в документации класса, что данные члены использовать и изменять крайне нежелательно для Вашего же блага. Python предполагает, что пользователи программных продуктов на этом языке будут людьми сознательными и грамотными, способными самостоятельно разобраться в нюансах кода и достаточно ответственными, чтобы не нарушать соглашения документации без объективных причин. Такой подход предоставляет ряд преимуществ, в частности даёт возможность пользователю более широкого использования приобретённых классов, а не только для решения узких задач, на которые их нацеливали авторы. В других языках использование закрытых членов класса невозможно без перепрограммирования кода источника с повторной компиляцией.

В следующих нескольких главах Вы сможете разобраться на практических примерах, как правильно создавать объекты. Вероятно, тогда смысл инкапсуляции станет для Вас более понятным. Чтобы спроектировать эффективный объект, необходимо хорошо разобраться в контексте его использования. Раскрытие смысла термина `контекст` мы также оставим на следующие главы.

С ключевой идеей, лежащей в основе полиморфизма, Вы также уже знакомы и неоднократно сталкивались с ней ранее. В Python каждый раз, когда нужно возвратить строковое представление переменной (например, для вывода на экран), можно воспользоваться функцией `str()`, символами обратного

ударения `` или применить спецификатор форматирования 4s. Все указанные способы посылают идентичное сообщение объекту переменной, в ответ на что этот объект реагирует соответствующим образом. В результате всегда возвращается строка символов. В этом состоит смысл полиморфизма: в ответ на одно и то же сообщение объекты разных типов возвращают разные результаты. При создании собственного класса необходим самостоятельно разработать методы, ответственные за возвращение ответа на родовые (т.е. характерные для данного класса) типы сообщений. В последующих главах, по мере того как будет возрастать уровень Вашего понимания объектов и его членов, мы рассмотрим разные способы решения этой задачи.

Я полагаю, что будет весьма поучительно сравнить следующие два листинга, в одном из которых определяется простой класс на языке C++ (листинг 9.3), а в другом – тот же класс на языке Python (листинг 9.4).

Листинг 9.3. Определение класса now (now.cpp) на языке C++

```
#include <stdio.h>
#include <time.h>
class now
{
public:
    time_t t;
    int year;
    int month;
    int day;
    int hour;
    int minute;
    int second;
    int dow;
    int doy;

    now()
    {
        time(&t);
        struct tm * ttime;
        ttime = localtime(&t);
        year = 1900 + ttime->tm_year;
        month = ttime->tm_mon;
        day = ttime->tm_mday;
        hour = ttime->tm_hour;
        minute = ttime->tm_min;
        second = ttime->tm_sec;
        dow = ttime->tm_wday;
        doy = ttime->tm_yday;
    }
};
```

```

    }
};

main (int argc, char **argv)
{
    now x ;
    fprintf ( stdout, "The year is %d\n", x.year );
}

```

Листинг 9.4. Определение класса now (now.py) на языке Python

```

#!C:\PYTHON\PYTHON.EXE
import time
class now:
    def __init__(self):
        self.t = time.time()
        self.storetime()

    def storetime(self):
        self.year, \
        self.month, \
        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst = time.localtime(self.t)

    def __str__(self):
        return time.ctime(self.t)

n = now()
print "The year is", n.year
print n
S='n'
print S

```

*Прим. В. Шипкова: просьба не обижаться любителям С++, но приведённые тексты программ - наглядно показывают, "ху из ху".

В следующей главе мы подробно, буквально "по косточкам", разберем всё, что происходит в программе now.py, но в данный момент нам достаточно только подчеркнуть наиболее характерные отличия.

Чтобы получить доступ из внешней программы к переменной `year`, в языке C++ её необходимо открыть с помощью ключевого слова `public`. В Python все члены класса являются общедоступными. (Эван Симпсон (Evan Simpson) по этому поводу сказал: "Python основан на доверии. В нём отсутствует механизм принудительного закрытия доступа для тех пользователей, которым в других языках программирования приходится тратить уйму своего времени, чтобы обойти эти запреты".) Другие ключевые слова C++ – `private` и `protected` – устанавливают соответственно закрытый и защищённый доступ к членам класса.

- • В C++ каждое выражение в программе должно заканчиваться точкой с запятой. В Python этого делать не надо.
- • В C++ программные блоки выделяются фигурными скобками {}, а в Python – отступами.
- • В C++ класс `now` имеет скрытый член под именем `this`, который могут видеть и произвольно использовать любые методы этого же класса. Чтобы упростить код программы, можно изменить строку `year = ttime->tm_year;` следующим образом: `"this->year = ttime->tm_year;`, а затем таким же образом поступить с остальными строками. В Python эквивалентом `this` является ключевое слово `self`, но его использование является обязательным. В следующей главе Вы узнаете, чем ещё особенным отличаются методы классов на языке Python.
- • В C++ (так же, как и в C) необходимо прибавить 1900 к значению года, возвращенному функцией `localtime()`. Python самостоятельно позаботится об этом.
- • В C++ для создания экземпляра класса `now` используется специальный метод `now()`, который называется конструктором класса. При реализации класса в Python вызывается специальный встроенный метод класса `__init__()`. В следующих главах мы познакомимся также с другими специальными методами классов, такими как `name`.

***Прим. В. Шипкова: в C++ имя конструктора совпадает с именем класса. Лично я считаю, что с методической точки зрения – это не верное решение. (а методику мне преподавали хорошо) ;-)**

Ещё одно отличие между определениями класса `now` в C++ и Python, которое, несомненно, первым бросилось Вам в глаза, – это то, что в Python классы определяются значительно проще. Примите к сведению также следующий факт: после того как Вы ввели текст программы `now.py`, Вы можете сразу же выполнить её. В C++ прежде чем выполнить программу `now.cpp`,

необходимо предварительно скомпилировать её. Если же у Вас отсутствует компилятор C++, то Вам придётся либо приобретать его коммерческую версию, либо загрузить достаточно массивный пакет файлов бесплатной версии компилятора C++ из Internet. Как говорит в таких случаях Франк Стаяно (Frank Stajano): "Python всегда поставляется в комплекте с заряженными батарейками".

***Прим. В. Шипкова: По всему Франк Стаяно – наш парень. :)**

Резюме

В этой главе Вы узнали, что такое объекты, что переменные в Python являются всего лишь только ссылками на объекты, какие базовые концепции лежат в основе объектно-ориентированного программирования.

В следующей главе мы создадим простой класс, детально рассмотрим его работу, а также научимся "общаться" с объектами этого класса.

Практикум

Вопросы и ответы

Приведите, пожалуйста, наглядный пример, раскрывающий суть полиморфизма.

Вы можете относиться к полиморфным объектам так же, как к стае котов. На ласку каждая особь может реагировать по-разному, но если швырнуть в них палку, реакция будет однозначной.

Наборы являются константными объектами и их нельзя изменять после создания. В то же время допускается, чтобы наборы содержали элементы разных типов, в том числе изменяемые объекты, например списки. Допускается ли в Python изменение списка, являющегося элементом набора?

Да. Объекты, входящие в состав наборов, полностью сохраняют свои первоначальные свойства. Набор не может быть изменен после того, как был создан, но Вы можете вносить любые изменения в список, помещённый в набор. Не допускается только удалять полностью весь список, но Вы можете удалить все элементы этого списка.

Если строки символов являются объектами, почему они не имеют свои собственные методы? Почему мы должны всегда импортировать модуль string?

В Python искусственно поддерживается различие между базовыми, типами данных и пользовательскими объектами. Если допустить, чтобы все объекты имели методы, то это приведёт к тому, что пользователи, кроме всего прочего, смогут добавлять или изменять методы всех объектов. Такой подход поднимает целый пласт проблем, над разрешением которых сейчас работает Guido с коллегами из SIG (Special Interest Group – группа по интересам).

Вопросы и ответы

1. Объекты должны иметь:
 - а) Состояние.
 - б) Функциональность.
 - в) Самих себя.
 - г) Тождественность.
2. Концепции объектно-ориентированного программирования основаны на следующих понятиях:
 - а) Предки, братья и сёстры, потомки, родственники по восходящей линии и просто объекты.
 - б) Классы, объекты, наследование, инкапсуляция и полиморфизм.
 - в) Время, пространство, деньги, стиль и класс.
 - г) Класс, поведение, состояние, тождественность и печенье.
3. Объекты являются:
 - а) Совокупностью вещей.
 - б) Реализациями классов.
 - в) Скорее котами, чем печеньем.
 - г) Откровениями Будды о природе программирования.

Ответы

1. а, б, и г. Объекты могут иметь состояние и/или функциональность, но также должны характеризоваться тождественностью.
2. б. Классы, объекты, наследование, инкапсуляция и полиморфизм – вот правильный ответ, хотя многие были бы не против включить в список и печенье.
3. а, б, в и г. Все ответы правильные. Если кто-то хочет возразить против откровений Будды, давайте поспорим.

Примеры и задания

Для ознакомления с более развёрнутым материалом по объектно-ориентированному программированию посетите раздел часто задаваемых вопросов на сервере <http://www.cyberdyne-object-sys.com/oofaq2/>. Не забывайте, что средства объектно-ориентированного программирования языка Python не отличаются такой же "закрытостью" и "засекреченностью", как в некоторых других языках.

Загляните на информационную страницу группы SIG по Python, расположенную по адресу <http://www.python.org/sigs/types-sig/>. Информация, представленная здесь, поможет Вам разобраться с сутью некоторых проблем, которые могут возникнуть в случае устранения различий между базовыми типами данных и объектами, определяемыми пользователем.

10-й час

Определение объектов

В этой главе Вы научитесь создавать и использовать в программах объекты. Вы узнаете, как послать сообщение нашему простейшему объекту, какова внутренняя структура объекта и для решения каких задач необходимы объекты. Закончив чтение данной главы, Вы сможете определять классы; создавать объекты классов; разрабатывать методы классов; наследовать классы.

Итак, продолжим рассмотрение класса `now`, который уже использовали в конце предыдущей главы.

Первый класс

В листинге 10.1 показан уже знакомый Вам код класса `now`.

Листинг 10.1. Класс `now`

```
class now:
def _init_(self):
    self.t = time.time()
    self.year, \
    self.month, \
    self.day, \
    self.hour, \
    self.minute, \
    self.second, \
    self.dow, \
    self.doy, \
```

```
self.dst = time.localtime(self.t)
```

При создании экземпляра класса в Python используется специальный метод `__init__()`.

Вызов этого метода выглядит так же, как обычная функция, за исключением того, что в списке параметров первым передаётся особый аргумент `self`, который используется при вызове всех методов класса. Этот особый аргумент является простой ссылкой на сам объект. Чтобы лучше понять смысл и назначение аргумента `self`, следует сделать небольшой экскурс по местам применения переменных и функций.

Как Вы помните, переменные создаются вместе с присвоением им значений. Именно это и происходит, когда мы вызываем метод `__init__()` для класса `now`:

```
self.t = time.time()
```

В этой инструкции явно создаётся переменная `t`, которой присваивается текущее значение машинного времени. Почему бы нам просто не создать переменную `t`, присвоить ей требуемое значение и использовать затем в программе обычным способом? Потому что в этом случае будет создана *локальная переменная*, а не *переменная-член* класса. Другими словами, если мы создаём переменную внутри функции, не определив её особым образом, то эта переменная будет видна только внутри функции. То же справедливо для класса. Так, если бы мы вместо конструкции с параметром `self` записали `t=time.time()`, переменная `t` была бы видима только внутри функции `__init__()`. Но эта переменная оставалась бы невидимой для всех объектов, создаваемых на базе класса `now`. Представьте себе, что функции связаны с закрытым чёрным ящиком. Чтобы поместить некую вещь внутрь ящика, нужно вызвать соответствующую функцию (или метод) и задать ей правильный набор аргументов. А чтобы взять что-то из ящика, нужно воспользоваться значением, возвращаемым этой функцией. Функции создают переменные внутри ящика, но для этого могут использоваться только значения, переданные с аргументами, и никакие другие внешние переменные. Для них внешний мир просто не существует. Хотя Вы можете вызывать методы и использовать возвращённые значения для присвоения другим переменным или создания новых переменных, сама функция не может ничего создавать вне пределов ящика, которому она назначена. Исключение из этого правила делается только для функций, использующих параметр `self`. Пример использования параметра `self` показан в листинге 10.1. *Переменные-члены* (терминология объектно-ориентированного программирования) в Python создаются так

же, как и обычные переменные, т. е. путём присвоения им значений. Это и происходит в листинге 10.1: `self.t`, `self.year`, `self.month` и т.д. – все это переменные, которые создаются как члены класса `now`.

Функция `localtime()` возвращает набор из 9 значений даты и времени. В рассмотренном выше листинге эти значения присваиваются отдельным переменным-членам. Как и при создании обычных переменных, имя переменной находится слева от оператора присвоения (`=`), а присваиваемое значение – справа. Переменной `dow` присваивается значение дня недели, `day` записывает номер года и `dst` принимает значение 1 для зимнего времени или 0 – для летнего.

После создания экземпляра класса `now` можно заглянуть из внешней программы внутрь объекта и просмотреть переменные, созданные функцией `__init__()`. Можно возвращать значения этих переменных, изменять их, добавлять новые методы и вызывать методы класса. Причем методы и переменные можно изменять в уже созданном экземпляре класса. Правда, в этом случае новые методы и переменные не будут автоматически переноситься в другие экземпляры класса, даже если они будут создаваться после внесения изменений в ранее созданный экземпляр. (В действительности существует способ модернизации класса в ходе выполнения программы, но эта тема выходит за рамки базового изучения Python.)

Как Вы видели в исходном коде класса `now`, показанного в листинге 9.4, создать экземпляр класса очень просто:

```
n=now()  
print "The year is", n.year
```

Создание экземпляра класса во многом напоминает вызов функции. В действительности так оно и есть. По данной команде Python скрытно выполняет ряд действий: создаётся родовой объект, который передаётся в функцию вместо аргумента `self`. Таким образом, создание экземпляра класса можно представить следующим образом:

```
n = now.__init__(объект)
```

После создания экземпляра класса появляется возможность напрямую обращаться к методу `__init__()`, как в следующем примере:

```
n=now()  
o=now.__init__(n)  
print "year", o.year, n is o
```


Если запустить на выполнение данный код, то появится сообщение об ошибке `AttributeError`, сообщающее Вам о том, что переменная `o` не является экземпляром класса `now` (пустой объект). Ошибка произошла из-за того, что в действительности аргумент `self`, переданный в функцию `__init__()`, не является ссылкой на родовой объект. Связь аргумента с объектом устанавливает Python, но только при соблюдении стандартного синтаксиса создания экземпляра класса.

Модернизация класса

Созданный нами класс `now` уже вполне применим на практике и эффективен, поскольку совмещает в одном объекте выполнение двух функций. Но, немного потрудившись, мы можем повысить эффективность нашего объекта. Этим мы сейчас и займёмся — создадим большой класс, который сможет делать для нас больше (извините за каламбур). Работу над классом мы начнём в этом разделе, затем продолжим в следующей главе и в главе, посвященной специальным методам классов Python.

Что должен делать наш новый класс? Ниже представлены очередные требования к классу:

- должен сообщать нам текущие дату и время;
- давать возможность устанавливать значение времени в экземпляре класса в допустимом для данной системы диапазоне;
- поддерживать простейшие арифметические действия с датами;
- должен быть легко расширяемым.

Со временем возникнут дополнительные требования, но пока ограничимся этим списком.

В листинге 10.2 показан слегка модифицированный класс `now`.

Листинг 10.2. Измененный класс `now`

```
#!C:\PYTHON\PYTHON.EXE
import time
class now:
    def __init__(self):
        self.t=time.time()
        self.storetime()

    def storetime(self):
        self.year, \
        self.month, \
```

```

        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst=time.localtime(self.t)

n=now()
print "The year is", n.year
print n

```

Все, что мы сделали, — это добавили в класс простой метод, с помощью которого в любой момент можно сохранять текущее время в соответствующей переменной члене. Это означает, что после создания экземпляра класса `now` можно в любой момент обновить в нем значение текущего времени:

```

n=now()
n.t=time.time()
n.storetime()

```

В альтернативном выполнении в переменной можно сохранять весь набор значений, возвращаемый локальной функцией `localtime()`. Затем можно создать методы для возвращения каждого отдельного значения по указанному имени. В листинге 10.3 показан фрагмент подобного выполнения класса, по которому Вы можете уяснить общий принцип решения.

Листинг 10.3. Альтернативный способ выполнения класса `now`

```

class now:
    def __init__(self):
        self.t=time.time()
        self.current=time.localtime(self.t)

    def year(self):
        return self.current[0]

    def month(self):
        return self.current[1]

```

Данный подход определения класса потребует, чтобы пользователь для возвращения требуемого значения вводил более пространные выражения. В то же время этот подход более традиционен для объектно-ориентированного программирования. Так, для возвращения года нужно использовать следующие строки:

```
n=now()  
print n.year()
```

*Прим. В. Шипкова: если приведённые методы добавить в выше приведённую программу, то при попытке выполнить команду `print n.year()` интерпретатор выругается, что тип `'int'` "невызываем". В чём тут фишка? А в том, что до этого была определена переменная класса, с таким же именем. Именно её Питон и юзает. По другому говоря, классический конфликт имён. Грамотно будет делать так:

- 1) процедуру обозначить `GetYear()`
- 2) названия методов должны обозначать действие, например:
 `GetYear():`
 `SetYear():`

Впрочем, об этом речь ещё будет.

Но Python не столь традиционен, как другие современные языки программирования.

При запуске программы `now.py`, код которой показан в листинге 10.2, при выполнении последней инструкции (`print n`) на экране появляется неожиданное сообщение:

```
<__main__.now instance at 7fa450>
```

Эта строка сообщает нам, что `n` является экземпляром класса. Но мы это и так знаем. Нам нужно иметь возможность сообщить Python, что именно должно выводиться при печати экземпляра класса. К счастью, в Python предусмотрен метод выполнения этой задачи. Для этого нам потребуется так называемый *специальный метод класса*. Изучению этой темы полностью будет посвящена глава 13. Сейчас мы только вкратце рассмотрим эти методы.

В действительности со специальными методами классов Вы уже встречались, например с `__init__()`. Каждый раз при создании экземпляра класса Python ищет данный метод в определении класса и автоматически выполняет его. Если данный метод явно не определён, ничего страшного не происходит. Хотя непонятно, в чём может состоять преимущество такого определения класса, когда не происходит никакой инициализации при создании экземпляра класса. В действительности такие классы применяются, но на более высоком уровне программирования для создания сложных объектных конструкций, рассмотрение которых выходит за рамки этой книги.

Когда встречается инструкция вроде `print n`, где `n` — экземпляр какого-либо класса, Python ищет в определении класса специальный метод `__str__()`. Этот метод требуется для возвращения строки, представляющей текущий экземпляр класса. Если явно не указано, что именно должно выводиться при печати экземпляра класса, то выводится простое стандартное описание экземпляра. В примере с нашим классом `now` импортированный модуль `time` предлагает стандартный набор функций для вывода требуемой информации. Для нас удобной будет функция `time.ctime()`. Измененный вариант класса `now` показан в листинге 10.4.

Листинг 10.4. Новые изменения в классе `now`

```
import time
class now:
    def __init__(self):
        self.t=time.time()
        self.storetime()

    def storetime(self):
        self.year, \
        self.month, \
        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst=time.localtime(self.t)

    def __str__(self) :
        return time.ctime(self.t)

n=now()
print "The year is", n.year
print n
print s
```

***Прим. В. Шипкова: последняя инструкция вызовет ошибку. `<s>` никак не объявлялась до попытки напечатать.**

Запустим теперь программу `now.py`. Результат показан на рис. 10.1 (безусловно, текущее время на моем и на Вашем компьютерах не совпадает).

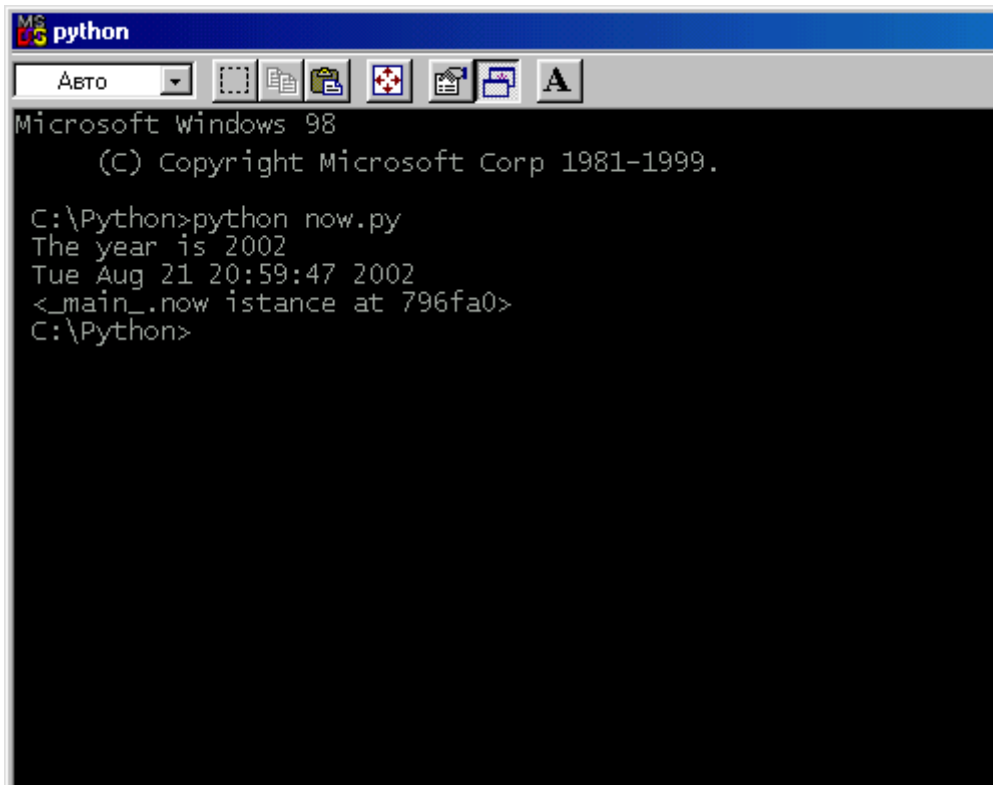


Рис. 10.1. Вывод измененной программы `now.py`

В инструкциях с `print` функцию `str()` можно выполнять явно:

```
n=now()  
s=str(n)  
print s
```

Для вывода строк в Python используется ещё один специальный метод `__repr__()`.

Экземпляр класса, выводимый этим методом, заключается между символами обратного удара (```). От метода `__str__()` данный метод отличается тем, что возвращаемая им строка затем может быть вновь преобразована в экземпляр того же класса. Например, строка, возвращённая функцией `__repr__(n)`, где `n` — экземпляр класса `now`, может быть преобразована обратно в экземпляр класса `now` с помощью какого-нибудь метода или функции, которые следует разработать отдельно. Позже мы рассмотрим это подробнее, а сейчас просто добавим в определение класса следующие строки:

```
def __repr__(self):  
    return time.ctime(self.t)
```

Как Вы видите, определение метода `__repr__()` следует тому же синтаксису, что и определение метода `__str__()`. Это

справедливо для большинства специальных методов классов Python. Но Python отчетливо различает методы `__str__()` и `__repr__()`, поэтому нам следует добавить в класс обе версии.

Производные класса

В листинге 10.5 показана текущая версия класса `now` с добавленными двумя методами вывода строк и рядом других изменений, которые мы сейчас рассмотрим.

Листинг 10.5. Класс `now` с методами `__str__()` и `__repr__()`:

```
#!C:\PYTHON\PYTHON.EXE
import time
class now:
    def __init__(self):
        self.t=time.time()
        self.storetimef)

    def storetime(self):
        self.year, \
        self.month, \
        self.day, \
        self.hour, \
        self.minute, \
        self.second, \
        self.dow, \
        self.doy, \
        self.dst=time.localtime(self.t)

    def __str__(self) :
        return time.ctime(self.t)

    def __repr__(self) :
        return time.ctime(self.t)

if __name__ == "__main__":
    n=now()
    print "The year is", n.year
    print n
    x=now()
    s='n'
    print s
```

Благодаря добавленной инструкции `if __name__...` мы можем теперь импортировать класс `now` в другие модули и в интерпретатор, как показано на рис. 10.2.

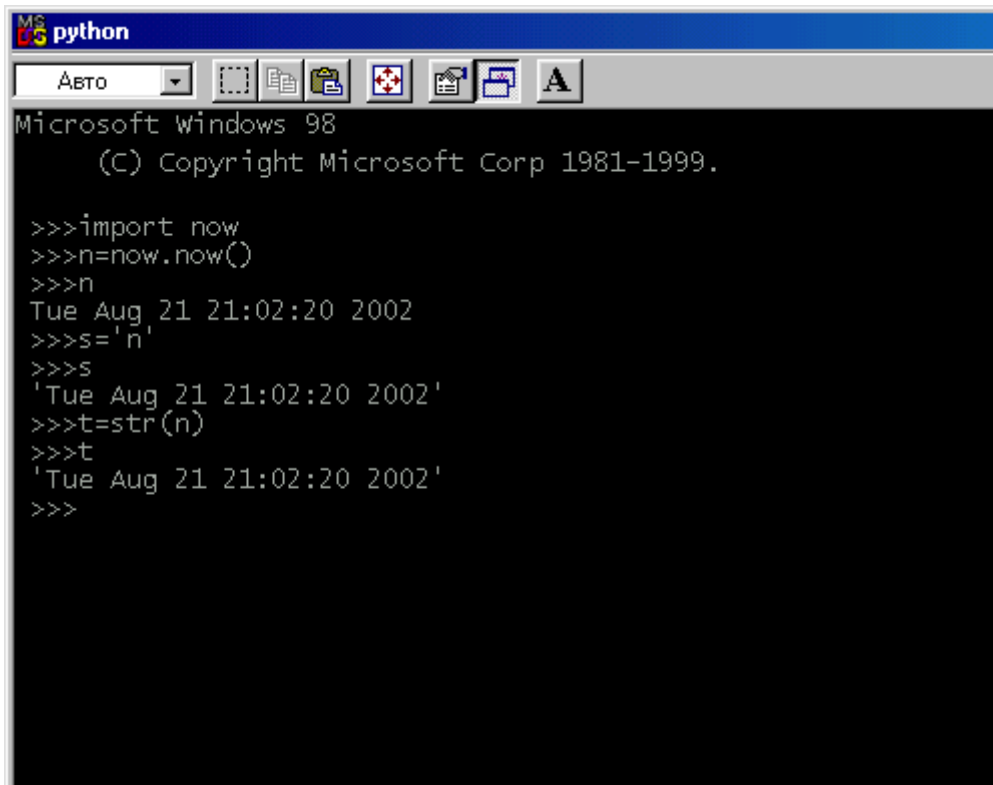


Рис. 10.2. Импорт класс `now` в интерпретатор

Мы можем использовать класс `now` в любом модуле. На данном этапе наш класс не отличается гибкостью, поскольку способен только создавать объект, содержащий поля даты и времени, которые были текущими в момент создания объекта.

Для повышения функциональности класса можно добавить в него другие специальные методы. Например, добавим следующие строки сразу после определения метода `__repr__()`:

```
def __call__(self, t=-1.0):  
    if t < 0.0:  
        self.t = time.time()  
    else:  
        self.t = t  
        self.storetime()
```

Этот новый метод позволяет делать следующее.

1. Обновлять текущее значение времени в экземпляре класса.
2. Устанавливать значение времени в экземпляре класса в допустимом для данной системы диапазоне (для UNIX в пределах 1970–2038 гг.).

Использование этих возможностей показано на рис. 10.3.



```
MS python
АВТО
Microsoft Windows 98
(C) Copyright Microsoft Corp 1981-1999.

>>>import now
>>>n=now.now()
>>>n
Tue Aug 21 21:02:20 2002
>>>n(0.0)
>>>n
Thu Jan 01 03:00:00 1970
>>>n(86400.0)
>>>n
Fri Jan 02 03:00:00 1970
>>>n(86400.0*365.25)
>>>n
Fri Jan 01 09:00:00 1971
>>>
```

Рис. 10.3. Выполнение класса `now` с методом `__call__()`

Метод `__call__()` позволяет вызывать экземпляр класса таким образом, как если бы это была обычная функция. Как видно из рис. 10.3, реализация объекта класса `now` всегда возвращает текущее время. Затем мы вызываем объект `n` и сообщаем ему (в параметре `t`) новое значение времени – `0.0`. Сообщение объекту `n` нулевого значения равносильно присвоению ему минимального значения времени в допустимом временном диапазоне – 1 января 1970 года, 00:00:00 часов. Но у меня на мониторе отобразилось время 03:00:00, поскольку на моём компьютере выбрана временная зона GMT+03:00 (Московское время), а для возвращения текущего времени используется функция `local_time()` из модуля `time`. Таким образом, время начала временного диапазона может меняться в зависимости от временной зоны и летнего-зимнего времени.

GMT означает *Greenwich mean time* (среднее время по Гринвичу). За нулевую была принята меридиана, проходящая через Королевскую обсерваторию в Гринвиче, Великобритания (*Royal Observatory at Greenwich*). От этой меридианы начинается отсчёт временных поясов во всем мире. Решение об этом было принято на Международной конференции по определению меридиан (*International Meridian Conference*) в 1884 г., и к нему присоединились все страны за исключением Франции. В 1970 г. состоялась новая конференция под эгидой Международного телекоммуникационного союза (*International Telecommunication Union*), на которой было принято решение

об отсчете по Гринвичу мирового времени – Coordinated Universal Time. Для мирового времени была выбрана аббревиатура UTC. (Эта аббревиатура никак не расшифровывается ни на одном языке. По идее она должна использоваться вместо GMT.)

Наследование класса

Созданный нами простой класс можно использовать как основу для создания новых классов, наследуя их от базового класса. Как Вы помните из предыдущей главы, наследование – это один из атрибутов объектно-ориентированного программирования. Теперь Вы готовы к использованию наследования на практике. В листинге 10.6 показано наследование класса `today` от класса `now`.

```
#!C:\PYTHON\PYTHON.EXE
import time
import now

class today(now.now):
    def __init__(self,y=1970):
        self.t=time.time()
        self.storetime()
if __name__=="__main__":
    n=today()
    print "The year is", n.year
    print n
    x=today()
    s='x'
    print s
```

При наследовании классов в Python для производного (дочернего) класса вызывается метод `__init__()`. При этом все методы, ранее определённые в базовом (родительском) классе, становятся доступными для нового производного класса точно так же, как если бы мы определили их в этом классе. С другой стороны, переменные-члены класса не будут появляться в производном классе до тех пор, пока мы явно не создадим их или пока не вызовем метод `__init__()` для базового класса. В листинге 10.6 первый метод нового класса явно создаёт переменную (см. строки 6,7). Данный код можно немного упростить, если вместо создания нового метода установить вызов метода `__init__()` базового класса в методе `__init__()` дочернего класса:

```
def __init__(self,y=1970):
    now.__init__(self)
```

Вызов метода `__init__()` базового класса `now` позволит избежать повторной записи того же кода в производном классе, что упростит код программы. Кроме того, в этом случае мы можем в любой момент изменить класс `now`, что повлечёт за собой автоматическое изменение всех модулей и классов, импортирующих или унаследовавших данный класс.

Прежде чем завершить эту главу, давайте добавим в класс `today` ещё один метод – `update()`, с помощью которого мы сможем устанавливать для объекта класса `today` значение времени в промежутке от 1970 до 2038 гг. Добавление метода `update()` показано в листинге 10.7.

Листинг 10.7. Добавление метода `update()` в класс `today`

```
#!C:\PYTHON\PYTHON.EXE
import time
import now
class today(now.now):
    def __init__(self,y=1970):
        now.__init__(self)

    def update(self,tt):
        if len(tt) < 9:
            raise TypeError
        if tt[0] < 1970 or tt[0] > 2038:
            raise OverflowError
        self.t = time.mktime(tt)
        self(self.t)

if __name__ == "__main__" :
    n = today ()
    print "The year is", n.year
    print n
    x = today()
    s = 'x'
    print s
    tt = (1999,7,16,12,59,59,0,0,-1)
    at.update (tt)
    print x
```

Новый метод `update` принимает набор значений даты и времени, выполняет простенькую проверку соответствия введённых данных допустимым параметрам и пытается с помощью функции `time.mktime()` извлечь из полученного набора значение времени. Время в Python отображается в виде числа с плавающей запятой. Время, заданное в нашем примере, выражается числом 932151599,0. Результат запуска программы `today.py` показан на рис. 10.4.

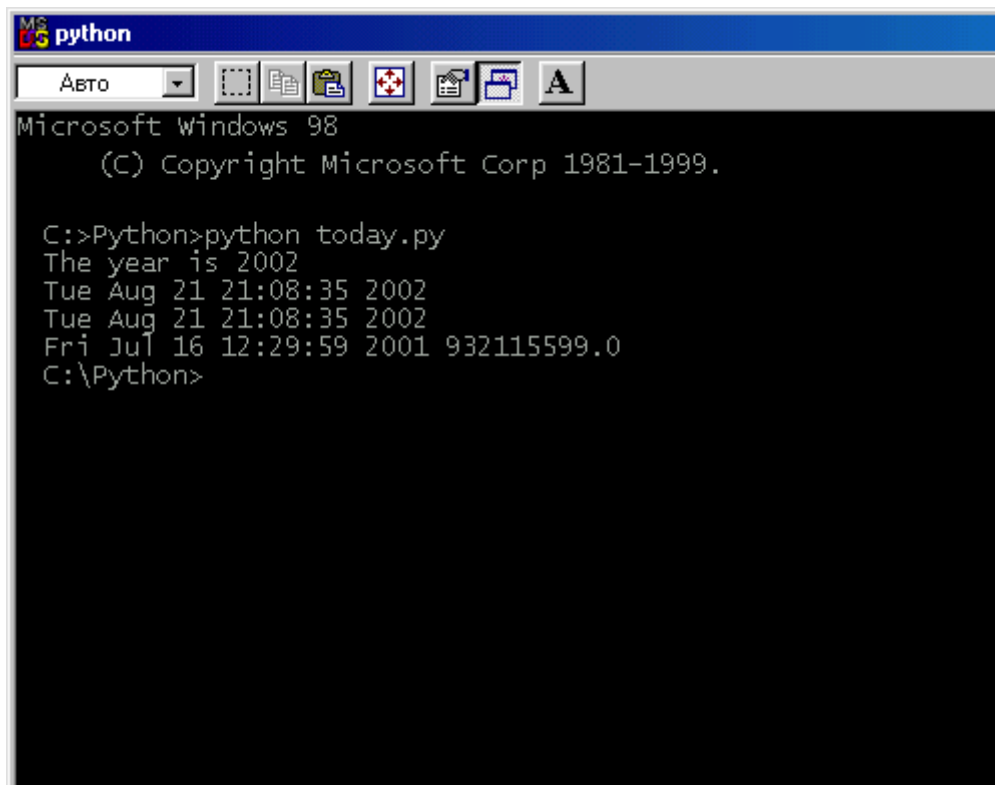


Рис. 10.4. Выполнение программы today.py

Следующую главу мы начнём с добавления новых методов в класс `today` для повышения его функциональности. Так, нам нужно будет иметь возможность складывать и отнимать значения времени.

Резюме

В этой главе Вы узнали, как создавать классы, объекты и методы классов, а также как наследовать классы.

Практикум

Вопросы и ответы

Почему 1970-й и 2038-й годы являются граничными в UNIX?

Дата 1 января 1970 года 00:00:00 выбрана как базовая в UNIX. Текущее время всегда сохраняется как целое число, имеющее размер 4 байта, или 32 бита. Один бит отводится на символ знака, а время представлено числом секунд от исходной даты. Таким образом, на запись числа секунд отводится 31 бит. В результате в большинстве систем UNIX дата 19 января 2038 года является предельной, так как исчерпывается запас битов и значение обнуляется. В некоторых версиях UNIX для представления даты используется беззнаковое целое число, что добавляет ещё один разряд. В результате предельная дата отодвигается на 68 лет. В других

версиях для представления времени отводится не 4, а 8 байт, что в принципе делает предельную границу недостижимой в реальном календаре.

11-й час

Концепции объектно-ориентированного программирования

В этой главе мы обсудим основные концепции объектно-ориентированного программирования, которое Вам будут крайне необходимы как для изучения материала данной книги, так и для выполнения будущих проектов. Мы рассмотрим следующие концепции: тождество, инкапсуляция, интерфейсы и соглашения; пространства имён и область видимости.

Тождество

Если мы представим себе числовой ряд, то не трудно заметить, что каждое число в нём обладает своими уникальными характеристиками. Как уже говорилось в главе 9, тот факт, что единица всегда равняется единице, но никогда не равна двум, лежит в основе свойства тождества объектов. Следует заметить, что свойство тождества объектов отличается по сути от математической операции тождества, которая определяется тем, что оставляет операнды неизменными. В случае с объектами ситуация не столь прозрачна, по крайней мере до тех пор, пока Вы не запомните, что в Python все цифры являются объектами. Любой числовой объект всегда легко отличить от нетождественного ему другого числового объекта.

Давайте проведём небольшой эксперимент и создадим простой объект в Python. Запустите программу, код которой приведён в листинге 11.1.

Листинг 11.1. Программа bunch.py

```
#!c:\python\python.exe
class bunch:
    def __init__(self):
        pass

if __name__ == "__main__":
    b = []
    for i in range (0,25):
        b.append (bunch())

    print b
```

Программа создаёт список экземпляров класса. При запуске программы следует обратить внимание на следующие моменты. Во-первых, список `b` содержит 25 экземпляров класса `bunch`. Во-вторых, любой экземпляр можно отличить от другого по двум параметрам: положению в списке и идентификационному номеру. Именно идентификационные номера объектов Python выведет при выполнении программы `bunch.py`, которые выглядят примерно так:

```
<__main__.bunch instance at 7f7f00>
```

Значение `7f7f00` является идентификационным номером объекта. Т. е., кто имеет опыт программирования на C, могут распознать в идентификационном номере адрес объекта или указатель на ячейки памяти, занимаемые объектом. Но в терминологии Python мы говорим об идентификационном номере объекта, который присваивается объекту при создании, чтобы иметь возможность отследить его и удалить в нужный момент.

Ещё Вы могли заметить, что все экземпляры класса `bunch` совершенно бессмысленны. Хотя мы чётко можем отличить любой экземпляр от соседних, между ними нет никакой разницы, поскольку для класса не определены ни методы, ни переменные-члены. Как плохие менеджеры, экземпляры класса `bunch` заняли свои ячейки памяти в ожидании того, когда компьютер от них избавится, ничего не делая все это время.

Код, показанный в листинге 11.2, делает практически то же самое, что и код из листинга 11.1, только вместо экземпляров класса `bunch` создаётся список экземпляров класса `today`, знакомого Вам по прошлой главе.

Листинг 11.2. Программа `today.py`

```
#!C:\python\python.exe
import today
if __name__ == "__main__":
    b = []
    for i in range (0,25):
        b.append(today.today()) ,
    print b
```

Если мы запустим на выполнение этот код, то убедимся, что экземпляры класса `today` сами по себе ничуть не функциональнее, чем экземпляры класса `bunch`. Чтобы добавить хоть немного функциональности, изменим инструкцию `b.append()` в цикле `for`, чтобы она выглядела следующим образом:

```
x = today.today()
x(i*86400)
b.append(x)
```

Правильный отступ сделайте самостоятельно. Затем запустите программу на выполнение. Вы увидите, что каждый экземпляр класса `today` представляет одно и то же время, но в разные дни. Теперь мы видим, что экземпляры действительно отличаются друг от друга. Различные экземпляры класса по крайней мере потенциально могут иметь какое-либо практическое применение. Как говорил Грегори Бейтсон (Gregory Bateson): "Информация – это разнообразие, которое порождает разнообразие". Наш модифицированный вариант программы `today.py` как раз несёт в себе разнообразие, порождающее разнообразие. Продвигаясь по списку объектов, мы можем проследивать, например, различия в высоте приливной волны каждый день в одно и то же время, вычислять вероятность солнечного затмения, среднее значение улова рыбы и др. И в этом состоит суть свойства тождества объектов. Нет смысла говорить о различии экземпляров класса, если эти различия не порождают новые различия.

Инкапсуляция

В главе 9 было дано определение инкапсуляции как концепции объектно-ориентированного программирования, подразумевающей объединение интерфейса и выполнения объектов. На примере класса `today` можно посмотреть, что в действительности подразумевается под этой концепцией. Что является интерфейсом нашего класса? Запомните, что интерфейс просто определяет способы вызова функций или методов или способ создания экземпляра объекта. Имея это в виду, давайте рассмотрим код класса, показанный в листинге 11.3.

Листинг 11.3. Класс `today`

```
class today(now.now):
    def __init__(self, y = 1970):
        now.now.__init__(self)

    def update(self, tt):
        if len(tt) < 9 :
            raise TypeError
        if tt[0] < 1970 or tt[0] > 2038:
            raise OverflowError
        self.t = time.mktime(tt)
        self(self.t)
```

К интерфейсу класса можно отнести две следующие инструкции:

```
__init__ (self, y = 1970)  
update (self, tt)
```

Эти два интерфейса можно представить как соглашение между программистом и пользователем. В соглашении говорится: если пользователь правильно активизирует интерфейс, то я, программист, обязуюсь, что пользователю будут возвращены данные, для получения которых был создан этот класс. В интерфейсе ничего не говорится о том, как должны выполняться функции-члены класса, подразумевается только, что они есть. Так, например, Вы можете разработать свою версию функции `time.mktime()` вместо представленной в листинге 11.3. Она может иначе работать и даже возвращать другие данные, только Вам нужно будет указать это в документации к своему классу.

В классе `now` использовались пять интерфейсов:

```
__init__ (self)  
storetime (self)  
__str__ (self)  
__repr__ (self)  
__call__ (self, t=-1.0)
```

Если быть точным, то в классе `today` можно насчитать семь интерфейсов, поскольку следует учесть все возможности получения доступа клиента к классу, включая предоставляемые Python возможности изменять как функциональность, так и состояние класса. Тут можно учесть даже саму переменную `t`, которая содержит значение времени как часть интерфейса класса. Правда, как правило, состояние класса обычно не должно изменяться пользователем, если только разработчик не снабдит пользователя инструкцией о том, как и зачем это можно делать. В отличие от других объектно-ориентированных языков программирования, в Python не предусмотрены внутренние механизмы, ограничивающие возможности изменения экземпляров класса внешними функциями. Для пользователя существует реальная возможность в ходе выполнения программы менять как состояние, так и функциональность любого объекта. Вы можете в документации рекомендовать пользователю не поступать так, но не можете это запретить.

Вы уже несколько раз читали в этой книге о важности составления правильной документации на методы и функции, но пока не знаете, как это можно сделать в Python. Любая функция, метод или модуль могут иметь встроенные строки документации, предназначенные для того, чтобы разъяснить

потенциальным пользователям назначение и принципы использования этого модуля. В листинге 11.4 показан модернизированный метод `storetime()` класса `now`.

Листинг 11.4. Модернизированный метод `storetime()`

```
def storetime(self):
    """
    storetime():
    Input: instance of class now
    Output: None
    Side effects: reads the member variable t and convert
    t into 9 TSember variables representing the same time
    as t.

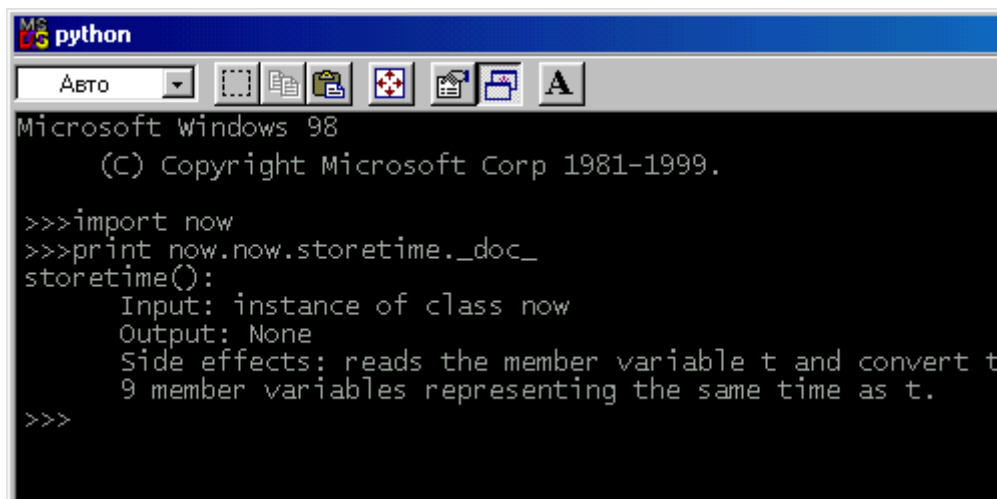
    """
    self.year, \
    self.month, \
    self.day, \
    self.hour, \
    self.minute, \
    self.second, \
    self.dow, \
    self.doy, \
    self.dst = time.localtime(self.t)
```

Текст, заключенный между тройными кавычками, объясняет, как работает метод `storetime()` и для чего его можно использовать. Ниже показан перевод строк документации на русский язык.

```
"""
storetime():
Ввод: экземпляр класса now
Вывод: ничего
Побочные эффекты: считывает переменную-член t и
преобразовывает t в 9 переменных-членов, представляющих
то же значение времени, что было в t.
"""
```

Вспомните, что все функции и методы в Python являются объектами. От обычных функций объекты отличаются тем, что характеризуются своим состоянием. Поэтому, когда в нашем примере мы добавляем строки текста, заключённые между тройными кавычками, в действительности мы иницилируем специальную встроенную переменную-член метода `storetime()`, которая носит имя `__doc__`. В результате документацию

функции или метода можно не только прочесть в программном коде, но и отобразить на экране, как показано на рис. 11.1.



```
MS python
АВТО
Microsoft Windows 98
(C) Copyright Microsoft Corp 1981-1999.

>>>import now
>>>print now.now.storetime.__doc_
storetime():
    Input: instance of class now
    Output: None
    Side effects: reads the member variable t and convert t
                 9 member variables representing the same time as t.
>>>
```

Рис. 11.1. Вывод на экран документации метода `storetime()`

Обратите внимание, что в документации не описывается, как именно метод `storetime()` выполняет поставленные перед ним задачи. Это сделано специально. Метод `storetime()` является частью интерфейса, договоренностью между пользователем и программистом о том что должен делать метод. При этом пользователя может совершенно не интересовать то, как метод реализует свою функциональность. Поэтому программист имеет полное право переписать по-своему функцию `mktime()` или вообще обойтись без неё, а все вычисления сделать в теле метода `storetime()`. Например, чтобы преобразовать значение времени, представленное в переменной `self.t`, в число дней, можно просто разделить `self.t` на 86'400 (число секунд в одном дне). Также можно вычислить остаток секунд после деления, если разделить `self.t` на 86'400 по модулю, а затем преобразовать этот остаток в значение времени. Затем, рассчитав число дней и зная текущую дату и время, не трудно вычислить дату события с указанием года, месяца, дня недели и т.д. В нашем примере было проще всего использовать стандартную функцию `mktime()` из модуля `time`, но Вы вольны подставить свою функцию, если она в большей степени отвечает вашим требованиям.

Но вернёмся к документации, так как это очень важное и полезное средство, предоставляемое Python. Имеет смысл снабжать документацией все модули вашей программы, особенно если предполагается, что с вашим кодом когда-либо будут работать другие люди. Вы сами обнаружите, насколько полезной бывает документация, когда в своей практике вернётесь к ранее написанному коду и попытаетесь его модернизировать, переписать или отладить.

Чтобы создать документацию для всего модуля, введите текст прямо в первой строке кода до инструкций и комментариев программы. Например, попробуйте ввести в файл приложения now.py следующую пробную строку документации:

```
"""This is a test documentation string for now
module....."""
```

Эту строку можно отобразить на экране командой `print now.__doc__`

Классы также могут иметь свою документацию в дополнение к переменным (или атрибутам) `__doc__`, связанным с каждым методом и функцией класса. Например, введём документацию класса `now`:

```
class now:
    """
    Эта строка показывает как можно добавить к классу
    документацию.
    """
```

Затем эту строку можно вывести следующей командой, предварительно импортировав файл `now`:

```
print now.now.__doc__
```

Первое `now` — это обращение к модулю, а следующее `now` — это имя класса, существующего внутри модуля. Переменная `__doc__` в данном случае является атрибутом класса `now`. По-моему, все очень просто.

Пространства имён и область видимости.

Где живут переменные, функции и методы, классы и модули? Мы можем исследовать этот вопрос с помощью строк документации, с которыми познакомились в предыдущем разделе. Начнём с того, что запустим наш интерпретатор Python в сеансе DOS (или в другой операционной оболочке). После приглашения `>>>` введите `dir()`. На экране появится сообщение

```
['_builtins__', '__doc__', '__name__']
```

На что это похоже? Не правда ли, это похоже на список?

Давайте поэкспериментируем:

```
>>> type(dir())
```

```
<type 'list'>
>>>
```

Действительно, перед нами список, но список чего? Чтобы ответить на этот вопрос, следует сначала обратиться к документации Python и посмотреть, для чего служит функция `dir()`: "При использовании без аргументов возвращает список имён в текущей локальной таблице символов. С аргументами — пытается вернуть список действительных атрибутов указанного объекта".

Атрибутами называются имена, такие как имя функции, переменной, класса и т.д. Давайте в нашем интерактивном сеансе работы с Python создадим переменную и ещё раз вызовем функцию `dir()`:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> spam = 42
>>> dir()
['__builtins__', '__doc__', '__name__', 'spam']
>>> spam
42
>>>
```

Функция `dir()` без аргументов возвращает список всех имён, известных Python в текущем сеансе работы. Всё вполне понятно, но что означает термин *текущая локальная таблица символов*? Чтобы разобраться с этим термином, обратимся ещё к одной встроенной функции — `locals()` — и посмотрим, что о ней говорится в документации: "Возвращает словарь, представляющий текущую локальную таблицу символов".

Таким образом, таблица символов — это словарь — один из базовых типов данных, представляющих имена (называемые ещё ключами), ассоциированные со значениями. Вызовем функцию `locals()` в нашем сеансе. Отобразится строка наподобие следующей:

```
>>> locals ()
{'spam': 42, '__doc__': None, '__name__': '__main__', '__builtins__':
<module 'builtin_'(built-in)>}
>>>
```

Вы видите, что только что созданная переменная `spam` представлена в списке и связана со значением 42. Обратите внимание, что ключ словаря `__builtins__` также отображён в списке и ему присвоено значение `<module 'builtin_'`

(built-in)>. Раз `__builtins__` является модулем, у него должны быть свои атрибуты, так же, как и у нашего текущего интерактивного сеанса. Мы можем отобразить список атрибутов модуля `__builtins__` следующим образом:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'FloatingPointError', 'IOError', 'ImportError',
'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'NotImplementedError', 'OSError', 'OverflowError',
'RuntimeError', 'StandardError', 'SyntaxError', 'SystemError', 'SystemExit',
'TypeError', 'ValueError', 'ZeroDivisionError', '__debug__', '__doc__',
'__import__', '__name__', 'abs', 'apply', 'buffer', 'callable', 'chr', 'cmp', 'coerce',
'compile', 'complex', 'delattr', 'dir', 'divmod', 'eval', 'execfile', 'exit', 'filter', 'float',
'getattr', 'globals', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
'issubclass', 'len', 'list', 'locals', 'long', 'map', 'max', 'min', 'oct', 'open', 'ord', 'pow',
'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round', 'setattr', 'slice', 'str',
'tuple', 'type', 'vars', 'xrange']
>>>
```

Итак, мы узнали следующее: `__builtins__` является модулем словаря (`type(__builtins__)`), который содержит таблицу символов (`dir(__builtins__)`) и переменные-члены `__doc__` и `__name__` с соответствующими записями; функция `dir()` возвращает все имена, хранящиеся в таблице символов, в числе которых переменные-члены `__doc__` и `__name__`.

Продолжая работу с Python в интерактивном режиме, импортируйте модуль `now` и повторите вызов `dir()`. Теперь модуль `now` также появился в списке, и если Вы введёте `type(now)`, то получите сообщение: `<type 'module'>` В действительности работа Python всегда заключена в какой-то модуль, независимо от того, работаете Вы в интерактивном режиме или запустили на выполнение файл программы. Все модули содержат атрибут `__name__`. Значением атрибута `now.__name__` будет, естественно, `now`. Если Вы дадите команду напечатать атрибут `__name__` сразу после вызова интерпретатора, то получите сообщение `__main__`. Между модулем `__main__` и всеми другими модулями существует одно отличие.

Это имя нельзя использовать как идентификатор. Так, если инструкция `now.__doc__` будет корректной для вызова документации модуля `now`, то документацию основного модуля Python нельзя вызывать командой `__main__.__doc__`. Вместо этого следует просто ввести в командной строке `__doc__`. Отсутствие идентификатора означает — вывести документацию текущего модуля `__main__`.

Аналогично, различие между вызовами функций `dir()` и `dir(now)` состоит в том, что во втором случае возвращается список действительных атрибутов указанного модуля `now`, который мы перед этим импортировали. В первом же случае по умолчанию будет отображён список атрибутов модуля `__main__`, который выглядит таким образом:

```
['__builtins__', '__doc__', '__name__']
```

Теперь импортируйте модуль `now` и вызовите функцию `dir` для него. Вы увидите следующее:

```
['__builtins__', '__doc__', '__file__', '__name__', 'now', 'time']
```

Атрибуты `__builtins__`, `__doc__` и `__name__` представлены в обоих модулях.

Теперь подытожим наши знания.

- При запуске Python создаётся модуль `__main__`, который содержит связанный с ним словарь (мы это видим при запуске функции `dir()`).
- Каждый раз при выполнении инструкции `import` модуль Python создаёт новый модуль, с которым связан собственный словарь (см. `dir(now)`).
- Следует отличать одноимённые атрибуты разных модулей.
- Чтобы обратиться к атрибуту добавленного модуля, нужно использовать имя этого модуля в качестве идентификатора, например, как `now.__doc__`. Для обращения к атрибутам основного модуля `__main__` идентификатор указывать не нужно.
- Имена переменных, функций и классов называются ключами или атрибутами. Все они представлены в словаре, ассоциированном с определённым модулем.
- Словарь, в пределах которого существуют атрибуты, в терминологии программирования называют *пространством имён*. (Вы, наверное, уже волновались, что мы никогда не дойдем до этого термина.)
- Встроенная функция `dir()` просто перечисляет все имена, определённые в указанном пространстве имён.

Каждый модуль имеет своё пространство имён и каждая функция имеет своё пространство имён. В Python все объекты также имеют свои пространства имён. Их нет только у базовых типов данных. Когда говорится, что Python оперирует несколькими пространствами имён, было бы правильнее сказать, что Python работает с несколькими уровнями пространств имён.

Словари пространств имён можно свободно просматривать. Вы увидите, что Python хранит список всех внешних загруженных модулей. В листинге 11.5 показан код, с помощью которого можно посмотреть этот список.

Листинг 11.5. Программа namespace.py

```
#!c:\python\python.exe
import sys
import now
k = sys.modules.keys()
print "Keys:", k
print "-----"
for i in k:
    if i == "__main__":
        print ">>>, i, "__dict__", sys.modules[i].__dict__
        print dir()
```

На печать программа выводит пространство имён только модуля `__main__`. Можно удалить строку 8 с инструкцией `if`, но в этом случае будет выведено слишком много информации, в которой потом будет трудно разобраться.

Функции также имеют свои пространства имён, но к ним нельзя получить доступ описанным выше способом. Код, показанный в листинге 11.5, позволяет просматривать список пространства имён любого модуля, но не функции. Чтобы получить доступ извне к пространству имён какой-либо функции, следует воспользоваться функцией `dir()`. Рассмотрим следующий код:

```
import sys
def f():
    """doc string"""
    z = 42
    print sys.modules["__main__"].__dict__["f"].__dict__
```

Хотя код выглядит логически правильно, если запустить его, Python покажет сообщение об ошибке `AttributeError`.

Но если заменить последнюю строку с `print sys.modules...` на следующие две строки:

```
print "DIR", dir(f)
print "DIR", dir(sys.modules["__main__"].__dict__["f"])
```

обе выведут один и тот же результат.

Но из функции `f()` запросто можно вывести список атрибутов, находящихся в поле зрения пространства имён функции. Изменим функцию `f()` следующим образом:

```
def f():  
    """строка документации"""  
    z = 42  
    print "f", dir()
```

Если мы теперь запустим функцию `f()` на выполнение и сравним результат с выводом функции `dir(f)`, то заметим существенную разницу. Список, выводимый при запуске функции `f()`, содержит единственное имя — `z`. В выводе функции `dir(f)` присутствовала ссылка на атрибут `__doc__`, но этот атрибут отсутствует в выводе `dir()` из тела функции `f()`.

Таким образом, функция не имеет доступа к своей собственной документации.

Функция также не может заглянуть в пространство имён другой функции. Введите в файл `namespace.py` определение функции `f()` и следующий код функции `d()`:

```
def d() :  
    """другая строка документации"""  
    z = 44  
    x = 9.999  
    print "d", dir()
```

Выполните затем обе функции. Вы увидите, что пространства имён этих функций различны:

```
f ['z']  
d ['X', 'Z']
```

Хотя в обоих пространствах имён представлены атрибуты под одним и тем же именем `z`, в действительности это две разные переменные.

Тем не менее функции могут видеть и изменять переменные, не относящиеся к их пространству имён. Создадим ещё одну функцию `e()`:

```
def e():  
    """другая строка документации"""  
    z = 22  
    global z  
    z = 9999
```

```
print "e", dir()
```

По определению функции можно предположить, что в ней существуют две переменные `z`: одна находится в локальном пространстве имён функции `e()`, другая — в пространстве имён модуля. Нам следует указать функции, какую переменную мы хотим изменить. Для этого используется ключевое слово `global`, которое говорит интерпретатору Python, что `z` лежит не в локальном пространстве имён, а во внешнем модуле. Но в Python нет ключевого слова `local`, которое бы имело противоположное действие. Поэтому, используя `global` для определения глобальной переменной, мы теряем одноименную локальную переменную в этой функции.

В Python можно выделить три уровня пространств имён.

- Пространство имён `__builtins__`. Хотя оно и открыто для всех модулей, функций и классов, но всегда остаётся прозрачным для них.
- Глобальное пространство имён. Это пространство имён принадлежит модулю. Хотя не существует функций, позволяющих заглянуть в пространства имён объектов модуля, тем не менее, мы можем просматривать, использовать и изменять атрибуты всех объектов с помощью идентификаторов.
- Локальное пространство имён. Это то, что видят классы, методы и функции.

Эта моя собственная классификация. В других книгах по Python Вы можете найти иные схемы организации пространств имён. Вы можете использовать в своей практике ту схему, которая Вам кажется более понятной.

Осталось ещё несколько моментов, которые Вам необходимо уяснить для полного представления о пространствах имён. Рассмотрим листинг 11.6.

Листинг 11.6. Программа `nesting.py`

```
#!C:\python\python.exe
import sys
global_var = 9999
def outer( )
    z = 42
    def inner():
        global z
        y = 666
        z = 99
    print dir()
```



```
inner()
print dir(), z
print dir()
outer()
print Z
```

Проанализировав данный листинг и результаты выполнения программы, мы можем сделать следующие выводы.

1. Возможно вложение функций. Можно создавать такие функции, которые существуют только внутри других функций.
2. Пространства имён не бывают вложенными. Внутренняя функция не видит переменные, существующие во внешней функции. С точки зрения пространства имён функции `inner()` и `outer()` никак не связаны, просто существуют внутри одного модуля.

Функция `inner()` не видит переменные, созданные в функции `outer()`. Инструкция `global z` сообщает функции `inner()`, что по ссылке на переменную `z` следует обращаться к глобальному пространству имён, игнорируя локальное. Ссылка на переменную `z` в строке 9 (`z=99`) не изменяет значение переменной `z` в функции `outer()`. Вместо этого создаётся новая глобальная переменная `z`.

Отношения между пространствами имён классов такие же, как между пространствами имён функций. Можно создавать вложенные классы, но их пространства имён будут совершенно независимыми. Каждый экземпляр класса владеет своим пространством имён и не видит атрибуты другого класса, даже если этот класс существует внутри самого объекта.

И последнее, что нам осталось сделать, — это обсудить вопрос видимости атрибутов. Во многих изданиях термины пространства имён и видимости атрибутов используются как взаимозаменяемые. Но это не совсем так. Под *видимостью* следует понимать как пространство имён, в котором существуют переменные, классы и функции, так и время существования объектов. Переменная, созданная внутри функции, характеризуется локальным пространством имён и локальной видимостью. По завершении функции все переменные, принадлежащие локальному пространству имён, выходят из поля зрения, т.е. удаляются из памяти. Python автоматически очищает память, занимаемую переменными, после их выхода за пределы видимости. Эта процедура осуществляется следующим образом. Python определяет пространство имён, вышедшее за пределы видимости, затем возвращает список всех ресурсов

данного пространства имён и последовательно удаляет их, после чего удаляет само пространство имён.

Резюме

В этой главе Вы познакомились с концепциями объекта, тождественности, инкапсуляции, пространства имён и видимости. В разделе, посвящённом тождественности, Вы узнали о том, что говорить о двух разных объектах можно только в том случае, если различия между ними имеют смысл. В разделе об инкапсуляции Вы узнали, что интерфейс представляет собой соглашение между пользователем и программистом о порядке использования класса. И, наконец, в последнем разделе Вы узнали, что список атрибутов, видимых объекту, а также видимость самого объекта контролируются пространствами имён.

Практикум

Вопросы и ответы

Мне казалось, что инкапсуляция — это способ защиты данных. Разве это не так?

Такое определение подойдёт для других языков программирования, но в Python под этим термином подразумевается исключительно выполнение объекта и способ обращения к нему. Плохо это или хорошо, но Python ориентирован на сознательность пользователей и на то, что они сами не станут нарушать соглашения с автором программного продукта и основные принципы объектно-ориентированного программирования.

Для чего нужно пространство имён `__builtins__`? Не проще ли копировать встроенные функции во все новые пространства имён?

Такое решение вполне возможно, но в этом случае в каждый объект добавлялось бы слишком много атрибутов общего назначения, многие из которых никогда не использовались бы объектом. Вместо этого в объекты передаются только ссылки на базовое пространство имён, что делает использование объектов более эффективным.

Контрольные вопросы

1. Что означает строка вывода `<__main__.bunch instance at 7f7f00>?`

а) Это сообщение об ошибке.

- б) Модуль `__main__` содержит экземпляр класса `bunch` с идентификационным номером `7f7f00`.
- в) Это пространство имён модуля `__main__`.
- г) Это свойство тождественности модуля `__main__`.
2. Какой синтаксис добавления документации является правильным?
- а) `docstring = "Documentation"`
- б) `__doc__ = "Documentation"`
- в) `"""Documentation1....."""`
- г) `doc = Documentation`
3. С помощью каких встроенных функций можно отобразить содержимое пространства имён?
- а) `namespace()`
- б) `dir(пространство_имён)`
- в) `locals()`, `globals()` и `dir(пространство_имён)`
- г) `scope()`

Ответы

1. б. Строка `<__main__.bunch instance at 7f 7f 00>` означает, что модуль `__main__` содержит экземпляр класса `bunch` с идентификационным номером `7f7f00`.
2. в. Строки документации должны быть заключены между тройными кавычками (одну строку можно просто взять в двойные кавычки) и размещены в начале кода модуля, класса, метода класса или функции.
3. б. Только функция `dir(пространство_имён)` возвращает список атрибутов указанного пространства имён, тогда как функции `locals()` и `globals()` возвращают словари, связанные с пространствами имён. Функция `dir()` напоминает встроенный метод словаря `keys()` с дополнительной сортировкой записей.

Примеры и задания

Создайте свою функцию `mktime()`. Возможно, Вам поможет в этом Web-страница *Peter Meyer's Calendar* (Календарь Питера Мейера), расположенная по адресу http://www2.papeterie.ch/serendipity/hermetic/cal_stud.htm.

Больше информации о концепциях объектно-ориентированного программирования можно узнать на Web-странице *High School Computing: The Inside Story* (Высшая школа программирования: взгляд изнутри) по адресу <http://www.inf-gr.htw->

zittau.de/~wagenkn/Natasha.Chen.html. Хотя Python в этой статье даже не упоминается, концепции объектно-ориентированного программирования раскрыты достаточно полно.

12-й час

Ещё о концепциях объектно-ориентированного программирования

В предыдущей главе мы обсудили важнейшие концепции объектно-ориентированного программирования: тождественность, инкапсуляцию и пространство имён. Теперь мы познакомимся с другими концепциями: защита данных с помощью закрытых переменных; полиморфизм; множественное наследование.

Защита данных с помощью закрытых переменных

В Python практически нет встроенных средств, отвечающих за защиту данных (другой термин — контролирующих доступ). В современных объектно-ориентированных языках программирования существуют явные способы контроля за доступом к членам классов. Так, например, в Java и C++ используются специальные ключевые слова, которые служат единственной цели — сообщают компилятору о том, является ли данная переменная или функция закрытой или открытой. Эти ключевые слова перечислены ниже.

- `public` (открытый);
- `private` (закрытый);
- `protected` (защищённый), отсутствие ключевого слова по умолчанию устанавливает в C++ закрытый доступ, а в Java — дружественный доступ.

Давайте сначала рассмотрим, для чего используются эти ключевые слова в других языках, чтобы потом лучше разобраться, как без них обходится Python.

- Ключевое слово `public` устанавливает открытый доступ. Любая функция или метод класса может свободно использовать и изменять любые переменные и функции, отмеченные ключевым словом `public`. Методы одного класса могут использовать и изменять переменные-члены другого класса, если они были открыты. Опасность, с точки зрения концепции ограничения доступа к данным, состоит в том, что случайное или целенаправленное изменение членов класса внешними функциями может привести объект к внутренним противоречиям.

- Ключевое слово `private` закрывает доступ. Только методы класса могут получать доступ к членам класса, определяющим его структуру и функциональность. Как правило, внешние функции не имеют доступа к переменным-членам класса. Преимущество этого средства состоит в том, что программист всегда знает, что происходит в классе, так как структура класса закрыта от случайного несанкционированного изменения внешними функциями.
- Ключевое слово `protected` защищает данные класса. Это означает, что доступ к защищённым переменным-членам могут получить только методы самого класса и методы всех производных классов. Защищённый метод может быть вызван только другим методом данного или производного класса. Преимущество этого средства, как и предыдущего, состоит в ограничении несанкционированного доступа к данным класса. Например, объект класса, контролирующего ввод-вывод данных на дисковод гибких дисков, было бы благоразумно защитить от поступления потоков данных с монитора.
- Дружественный доступ в Java означает, что любой класс, происходящий из файла в текущем каталоге как целевой, может изменять структуру и функциональность дружественного класса, а также использовать его методы.

В Python нет никаких специальных ключевых слов, контролирующих доступ к членам класса. Как правило, доступ к данным определяется главным образом соглашениями, а также (в меньшей степени) средствами именования классов, переменных и методов. Имеется в виду использование символов подчёркивания перед и после имени для указания того, что в Python не предполагается безрассудное использование этих данных любыми классами, методами и функциями. Существует ограниченный список зарезервированных ключевых слов, начинающихся с символов подчёркивания, которые мы рассмотрим ниже. Но использование символов подчёркивания также является всего лишь видом соглашения, так как не существует никаких механизмов, ограничивающих доступ к этим переменным и функциям. Любой программист может свободно получать доступ и изменять любую реально существующую переменную.

***Прим. В. Шипкова: закрытие и защита членов полезная вещь только для зарабатывания денег на себе подобных, и близких своих, что не является этичным. Личное мнение. Что касается целостности исходного кода, я думаю, мало найдётся идиотов от программирования, которые захотели бы вырыть себе яму. ;) Кроме того, существуют средства цифровой подписи, такие как OpenPGP, позволяющие контролировать тексты скриптов, с точностью до бита. (т.е. точнее некуда).**

Первый специальный символ — это символ подчёркивания `"_"`. Он используется только при запуске интерпретатора Python в интерактивном режиме. При вводе любого выражения, такого как `1+1`, Python сохраняет результат в промежуточной переменной под именем, которая создаётся только после ввода выражения. Символ может использоваться пользователем как имя только в фоновом режиме работы, но лучше это имя вообще не использовать.

***Прим. В. Шипкова: ума не приложу, кому придёт в голову, обзывать переменную символом подчёркивания. ;)**

Другое специальное имя, точнее группа имён, — это любое имя, начинающееся с одного символа подчёркивания. На эти имена нужно обратить особое внимание. Любое имя, начинающееся с одного символа подчёркивания, не импортируется в пространство имён вместе с модулем при использовании выражения `from <module> import *`. Чтобы лучше понять суть сказанного, давайте рассмотрим листинг 12.1.

Листинг 12.1. Программа `imptest1.py`

```
#!C:\python\python.exe
_CHANGEOVER="Sep 3 1752"
from today import *
```

Это очень простая программа. Всё, что она делает, — это создаёт переменную, которая видна только в файле `imptest1.py`. В листинге 12.2 показана попытка использовать эту переменную.

Листинг 12.2. Программа `imptest2.py`

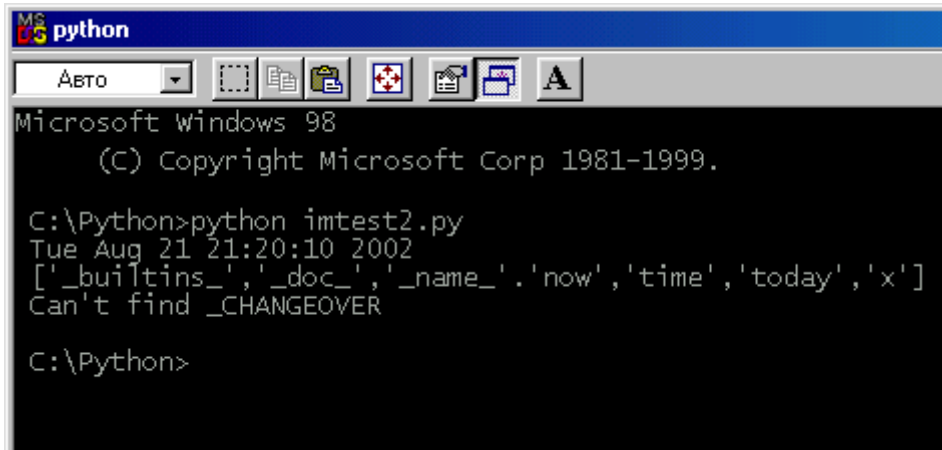
```
#!c: \python\python.exe
from imptest1 import *
x = today()
print x
print dir()
try:
    print _CHANGEOVER
except:
    try:
        print imptest1._CHANGEOVER
    except:
        print "Can't find _CHANGEOVER"
```

***Прим. В. Шипкова: пример несколько замудрён. Я бы сделал**

несколько проще.

Если Вы забыли, как работают инструкции `try` и `except`, перечитайте главу 8. Вывод программы `imptest2.py` показан на рис. 12.1.

Функция `dir()` вывела список атрибутов пространства имён модуля `imptest2.py`, но в нём отсутствует переменная с именем `_CHANGEOVER`. В программе `imptest1.py` она была доступна, но теперь исчезла.



```
MS python
Авто
Microsoft Windows 98
(C) Copyright Microsoft Corp 1981-1999.

C:\Python>python imptest2.py
Tue Aug 21 21:20:10 2002
['_builtins_', '_doc_', '_name_', 'now', 'time', 'today', 'x']
Can't find _CHANGEOVER

C:\Python>
```

Рис. 12.1. Выполнение программы `imptest2.py`

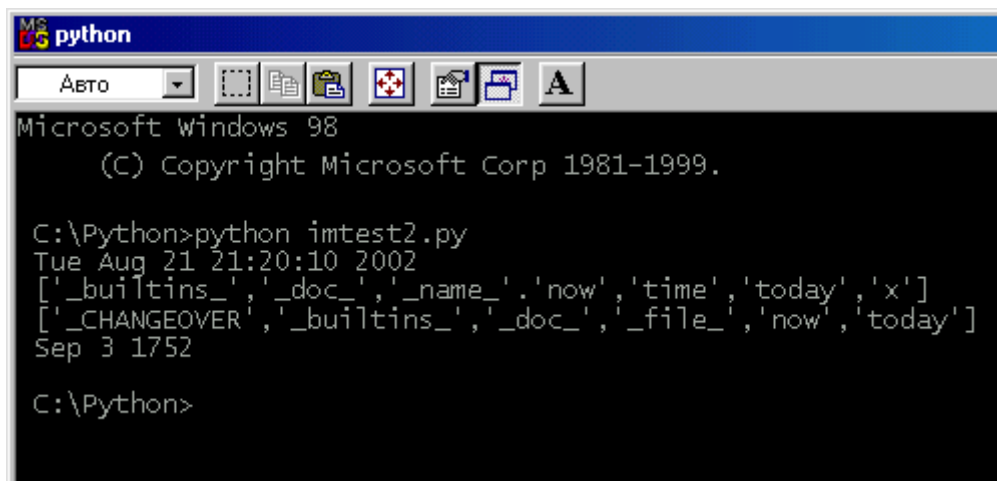
Чтобы получить доступ к этой переменной, нужно изменить код программы, как показано в листинге 12.3.

Листинг 12.3. Измененная программа `imptest2.py`

```
#!c:\python\python.exe
import imptest1
x = imptest1.today()
print x
print dir()
print dir(imptest1)
try:
    print _CHANGEOVER
except:
    try:
        print imptest1._CHANGEOVER
    except:
        print "Can't find _CHANGEOVER"
```

***Прим. В. Шипкова: как и предыдущая программа, эта могла бы быть и попроще.**

Результат выполнения измененной программы `imptest2.py` показан на рис. 12.2.



```
MS python
АВТО
Microsoft Windows 98
(C) Copyright Microsoft Corp 1981-1999.

C:\Python>python imptest2.py
Tue Aug 21 21:20:10 2002
['_builtins_', '_doc_', '_name_', 'now', 'time', 'today', 'x']
['_CHANGEOVER', '_builtins_', '_doc_', '_file_', 'now', 'today']
Sep 3 1752

C:\Python>
```

Рис. 12.2. Выполнение измененной программы `imptest2.py`

Как Вы видите, если избежать использования инструкции `from <module> import *`, то с доступом к переменной `_CHANGEOVER` не возникнет никаких проблем.

***Прим. В. Шипкова: я бы вообще рекомендовал избегать подобных вариантов. И пространство имён засоряется (влияет на скорость работы), и конфликт может вызвать по именам (маловероятно, но тем не менее, следует стремиться к простоте – первому признаку гениальности).**

Следующая группа специальных имён начинается и заканчивается двумя символами подчёркивания. Сюда относятся специальные методы классов, о которых мы поговорим в следующей главе. С некоторыми из них Вы уже встречались:

`__call__`, `__init__` и др.

Определения всех этих методов встроены в Python, но чтобы воспользоваться их функциональностью, нужно самостоятельно разработать выполнение требуемых методов в своей программе.

Наконец, последняя группа содержит имена, которые Вам вряд ли потребуются, по крайней мере в начале вашей практики. Рассмотрим их только для того, чтобы завершить тему. Методы и переменные-члены класса, чьи имена начинаются, но не заканчиваются двумя символами подчёркивания, называются *ограниченными* (*mangling* – этот термин был позаимствован из C++). Идея состоит в том, что если программист не хочет, чтобы пользователи класса видели некоторые переменные или методы, то перед их именем можно поставить два символа подчёркивания. Чтобы увидеть эти

методы или переменные, пользователь должен воспользоваться "специальными средствами", о которых можно узнать из документации, представленной на домашней Web-странице Python по адресу <http://www.python.org/>.

В следующем разделе мы поговорим о том, как в Python реализована концепция полиморфизма.

Полиморфизм

В главе 9 было дано определение полиморфизма как возможности передачи одного сообщения объектам разных типов с получением разных ответов от них. Примером, полиморфизма может быть использование метода `__init__()` в классах `now` и `today`, которые мы рассматривали в предыдущих главах. Python посылает сообщения классу при создании его экземпляра. Программист при создании класса в Python должен написать выполнение метода `__init__()`, чтобы иметь возможность контролировать процесс создания объекта. Впрочем, в некоторых случаях нет необходимости в методе `__init__()`, например, когда нужен объект с определённой функциональностью, а его состояние не имеет значения. Таким образом, полиморфизм не бывает встроенным. Программисту следует приложить определённые усилия, чтобы объект отвечал на сообщения особым образом.

Что касается метода `__init__()`, то программист волен изменять его для каждого класса без всяких ограничений, создавая для класса уникальный набор свойств. Можно изменять число аргументов, принимаемых методом `__init__()`, за исключением первого. Все специальные методы классов требуют, чтобы первым аргументом был экземпляр класса. Поэтому, по соглашению, первый аргумент всегда называется `self`. Опять-таки, это только соглашение. В Python нет никаких механизмов, отслеживающих имя первого аргумента. Но профессиональный программист никогда не станет его менять. Следование соглашениям делает код программы более читабельным как для других программистов, так и для Вас.

***Прим. В. Шипкова: также нельзя забывать про некоторое количество документации, но не столько, чтобы код в ней растворялся. Если требуется много документации - следует создать отдельный справочник.**

Метод `__init__()` не возвращает никаких значений. Если по какой-либо причине инициализация объекта сорвалась, должен быть предусмотрен запуск исключения (английский термин "*throw and exception*" можно перевести как "метнуть исключение"). Обычно в этом случае запускается исключение

`AttributeError`, хотя Вы можете разработать своё более подробное исключение с указанием того, при создании объекта какого класса произошел сбой. Это также хороший пример полиморфизма. Исключения также являются классами, поэтому не составит особого труда создать своё новое исключение, унаследовав его от стандартного.

Иногда при разработке полиморфных объектов следует учитывать необходимость поддержания стандартного интерфейса. (Вспомните концепцию инкапсуляции, которую мы рассматривали в прошлой главе.) Например, метод `__add__()`, который используется для сложения двух объектов с помощью оператора `+`, обязательно требует наличия двух аргументов — `self` и `other` -- и всегда возвращает новый экземпляр класса. Мы рассмотрим этот метод подробнее в следующей главе.

Хотя Python предоставляет большой набор специальных полиморфных методов, тем не менее, как Вы увидите в следующей главе, с их помощью нельзя удовлетворить все требования пользователей к объектам. Поэтому, как правило, создание объекта не обходится без написания собственных методов, придающих объекту особую функциональность. С помощью наследования не сложно создать целую иерархию объектов, отвечающих по-разному на одно и то же сообщение. Например, исчисления дат и цифр у индейцев племени Майя во многом совпадали, хотя имели свои особенности. Так, для цифр использовалась система счисления с основанием 20, для дат использовалась смешанная система счисления с основанием 20 и для цифр во второй позиции — с основанием 18. При создании программы, воспроизводящей данные системы счисления, класс дат можно унаследовать от класса цифр, сохранив в целом функциональность класса и полностью сохранив состояние. Иногда приходится выполнять конвертацию данных между классами. С этой целью проще всего будет снабдить класс цифр методом `convert()`, который будет автоматически возвращать дату. Класс дат, унаследованный от класса цифр, будет иметь одноименный метод, только в этот раз преобразующий дату в число. Для преобразования достаточно послать одно и то же сообщение экземплярам любого класса (`<экземпляр>.convert()`), но каждый раз будет выполняться то преобразование, которое соответствует типу объекта.

В следующем разделе мы вновь поговорим о наследовании классов, но в этот раз о том, как один класс унаследовать от нескольких родительских классов.

Множественное наследование



В главе 10 Вы узнали о том, как наследуются классы. В листинге 10.5 было показано, как класс `today` наследуется от класса `now`. В этом разделе Вы узнаете, как выполнять множественное наследование, т.е. как создавать класс, наследующий состояние и функциональность от нескольких родительских классов.

До сих пор множественное наследование применялось главным образом для создания так называемых *смешанных* классов. Существуют классы, содержащие настолько общие методы, что их можно объединить практически с любым другим классом. Например, можно создать класс, методы которого будут просматривать пространства имён указанных объектов и возвращать список строк, представляющих все атрибуты объекта. Такой класс можно объединить с любым другим классом Вашего приложения. В результате, вместо того чтобы создавать метод `__repr__()` для каждого нового класса, Вы сможете использовать унаследованный метод, тем самым повысив эффективность программирования.

Поскольку наш класс `today` уже содержит метод `__repr__()`, мы прибавим к нему другой родительский класс. Вам, наверное, приходилось в жизни сталкиваться с римскими цифрами, хотя бы на циферблатах некоторых часов. Давайте создадим класс, который будет представлять годы в виде римских цифр. (Преобразовывать десятичные значения в римские значительно проще, чем наоборот.) В табл. 12.1 показаны символы, используемые в римских цифрах (в компьютерной терминологии их следует называть *литералами*).

В информатике *литералами* называют минимальные порции данных (буквы-кванты), передающиеся в интерпретатор (*parsing engine*). В качестве интерпретатора может выступать программа или её часть, которая считывает введённый структурированный пакет данных и определяет его смысл, анализируя порядок литералов в этом пакете. Даже отдельные литералы могут иметь свой смысл (например, символы `"\"` и `"|"` в командной строке DOS). Чем больше литералов в строке передаётся интерпретатору, тем больше последовательных команд они содержат. Главное, чтобы после выполнения всех этих команд Вы чётко представляли себе, что произошло.

Таблица 12.1. Символы (или литералы), используемые в римских цифрах

Символ	Смысл
I	1
V	5
X	10

L	50
C	100
D	500
M	1000

Между прочим, это не полный список. Монахи в средневековье значительно доработали систему римских цифр, добавив не только новые символы, но и систему модификаторов, выступающих в роли коэффициентов умножения. Так, символ X с черточкой над ним означал 10000, т.е. черточка сверху означала умножение на 1000. Но нам в нашей программе не потребуются такие сложности. В листинге 12.4 показан простой смешанный класс, который преобразовывает годы в римские цифры.

Листинг 12.4. Программа roman.py

```
!c:\python\python.exe
import string
import sys

class roman:
    def __init__(self,y):
        if y < 1:
            raise ValueError
        self.rlist = []
        ms = y / 1000
        tmp = y % 1000
        if ms > 0:
            self.rlist.append("M" * ms)

        ds = tmp / 500
        tmp = tmp % 500
        if ds > 0:
            self.rlist.append("D" * ds)

        cs = tmp / 100
        tmp = tmp % 100
        if cs > 0:
            self.rlist.append("C" * cs)

        ls = tmp / 50
        tmp = tmp % 50
        if ls > 0:
            self.rlist.append("L" * ls)

        xs = tmp / 10
        tmp = tmp % 10
```

```

if xs > 0:
    self.rlist.append("X" * xs)

vs = tmp / 5
tmp = tmp % 5
if vs > 0:
    self.rlist.append("V" * vs)

js = tmp
if js > 0:
    self.rlist.append("I" * js)

def ryear(self):
    s = ""
    for i in self.rlist:
        s = s + i
    return s
def __repr__(self):
    return(self.ryear())

if __name__ == "__main__":
    if len(sys.argv) > 1:
        yr = string.atoi(sys.argv[1])
    else:
        yr = 1999
    x = roman(yr)
    print x.ryear()

```

***Прим. В. Шипкова:** то что, блоки условий никак не выделяются, не супер. Жаркие споры не утихают на протяжении уже более 7 лет – надо ли это делать. Если бы такая возможность была – я был бы не против (язык от этого хуже не станет). Такое решение сняло бы часть ответственности с программиста, и переложило бы её на плечи интерпретатора. И это тоже не так плохо. С другой стороны, стиль Питона тогда, стремительно покотился бы к C++, с его шаманизмом. Кроме того, стиль с явным выделением, расслабляет программиста, и ухудшает читабельность. Что является довольно серьёзными отрицательными моментами.

Блоки, завершающиеся функцией `append()`, представляют собой простейший способ подбора необходимого числа символов. Мы воспользовались свойством оператора умножения (*) распознавать в операндах строки и преобразовывать выражение умножения строки на число в повторение данной строки указанное число раз. Таким образом, мы составляем строку римской цифры путём конкатенации соответствующего числа символов тысяч, сотен и т.д. Например, 400-м годам

будет соответствовать выражение "C" * 4, которое возвратит строку "CCCC".

Запуск программы `roman.py` без аргумента возвратит 1999 г. в виде римской цифры. Если необходимо преобразовать другой год, введите его в качестве аргумента. На рис. 12.3 показана работа программы без аргумента и с аргументом – 2000 г.

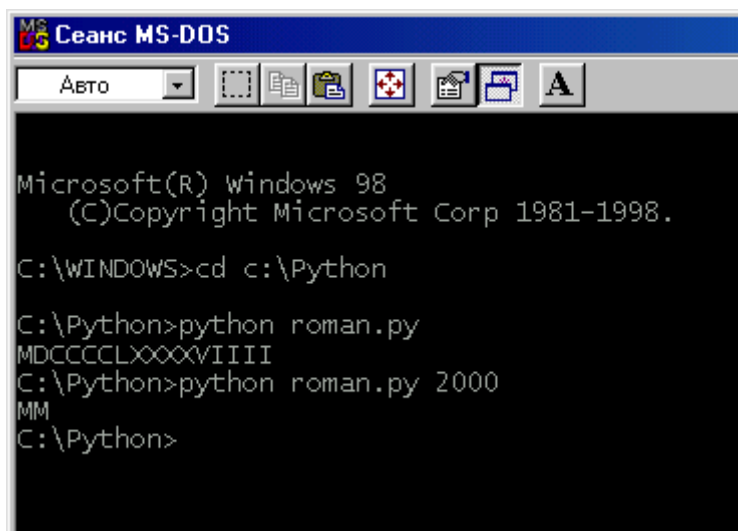


Рис. 12.3. Выполнение программы `roman.py`

Если знатоки римских цифр посмотрят на результаты выполнения нашей программы, они будут долго смеяться. Хотя иногда можно встретить написание девятнадцатого столетия цифрой `MDCCCC`, гораздо чаще встречается запись `MCM`. И уж точно, Вам никогда не повстречается число 90 в виде `LXXXX`, а только как `XC`, точно так же, как и 9 будет выражаться цифрой `IX`, а не `VIIII`. Следуя соглашениям римского исчисления, 1999 г. следовало бы записать как `MCMXCIX`. Так что наш класс `roman` годится для демонстрационных целей, но не для практического использования. Мы могли бы создать класс `roman` иначе без переменных-членов, оставив в нём только методы. Весь код преобразования десятичных цифр в римские можно вынести из метода `__init__()` в отдельный метод. Тогда все классы, произведенные от `roman`, унаследуют этот метод и смогут использовать его без обращения к методу `__init__()`. Ещё лучше будет смешать класс `roman` с другим классом, используя множественное наследование. Пример создания класса `today` путём множественного наследования от классов `now` и `roman` показан в листинге 12.5.

Листинг 12.5. Смешанный класс `today-roman.py`

```
#!c:\python\python.exe
#!/usr/local/bin/python
```

```

import time
import now
import roman

#class today(now.now,roman.roman):
class today(roman.roman,now.now):
    def __init__(self, y = 1970):
        now.now.__init__(self)
        roman.roman.__init__(self,y)
    def update(self,tt):
        if len(tt) < 9 :
            raise TypeError
        if tt[0] < 1970 or tt[0] > 2038:
            raise OverflowError
        self.t = time.mktime(tt)
        self(self.t)
        roman.roman.__init__(self,self.year)

if __name__ == "__main__":
    n = today()
    print "The year is", n.year
    print n
    x = today()
    s = `x`
    print s

    tt = (1999,7,16,12,59,59,0,0,-1)
    x.update(tt)
    print x, x.t
    print "Roman", x.ryear()
    st = `x`
    print st

```

На рис. 12.4 показан результат выполнения этой программы.

```
Microsoft(R) Windows 98
(C)Copyright Microsoft Corp 1981-1998.

C:\Python>python today-roman.py
The year is 2002
Tue Aug 21 21:27:32 2002
Tue Aug 21 21:27:32 2002
Fri Jul 16 12:59:59 2001 932115599.0
Roman MDCCCCLXXXVIII
Fri Jul 16 12:59:59 2002
C:\Python>
```

Рис. 12.4. Выполнение программы today-roman.py

Давайте добавим в класс `roman` метод `__repr__()`. Для этого добавьте в файл `roman.py` следующие строки сразу после метода `__repr__()`:

```
def __repr__(self):
    return(self.ryear())
```

Затем вновь запустите программу `today-roman.py`. В выводе программы ничего не изменится. Теперь изменим порядок наследования класса `today` с `class today(now.now, roman.roman):` на `class today(roman.roman, now.now):` и вновь запустим программу `today-roman.py`. В этот раз вывод программы изменился, как показано на рис. 12.5.

```
Microsoft(R) Windows 98
(C)Copyright Microsoft Corp 1981-1998.

C:\Python>python today-roman.py
The year is 2002
Tue Aug 21 21:27:32 2002
MDCCCCLXX
Fri Jul 16 12:37:58 2002
Roman MDCCCCLXXXVIII
MDCCCCLXXXVIII
C:\Python>
```

Рис. 12.5. Результат изменения порядка наследования класса today

В первом варианте программы существовал только метод `__repr__()`, унаследованный от класса `pow`. Но когда мы изменили порядок наследования классов, активным стал метод `__repr__()`, унаследованный от класса `roman`. На этом примере мы видим, насколько важным может оказаться правильный порядок наследования класса. Это общий принцип множественного наследования: если в родительских классах определены одноимённые методы, то в дочернем классе сохраняется только тот вариант метода, который был унаследован первым (в списке родительских классов был левее). В противном случае внутри производного класса могли бы возникнуть противоречия. В результате неправильной установки очередности наследования может оказаться, что некоторые функции в программе работают не так, как ожидалось.

Резюме

В этой главе Вы узнали, что хотя в Python нет специальных средств, обеспечивающих защиту данных, существуют определённые соглашения, следование которым позволит предохранить классы и переменные от неправильного использования или даже скрыть их. Вы узнали дополнительные сведения о полиморфизме и о том, как использовать множественное наследование для создания смешанных классов. В следующей главе мы поговорим подробнее о специальных методах классов, которые Вам уже встречались в предыдущих главах.

Практикум

Вопросы и ответы

Не станет ли программный код слишком запутанным после применения множественного наследования?

Ничуть! Особенно если учесть, насколько эффективнее может стать Ваш код после применения множественного наследования к большой группе взаимосвязанных классов. Тем самым Вы сможете значительно сократить длину кода, удалив из него повторяющиеся блоки, что только улучшит читабельность.

Мы начали с календаря Майя, а затем перескочили на римскую систему исчисления. А как же календарь Майя?

Не беспокойтесь, мы к нему ещё вернёмся. А пока, в качестве домашнего задания, попробуйте усовершенствовать нашу программу, чтобы она возвращала правильные римские цифры.

Чем отличается множественное наследование в Python от других языков программирования?

Концепция та же, но синтаксис в других языках обычно более сложный. Так, в языке C++ возможно создание такого хитросплетения наследований, что по сложности оно превзойдёт старые линейные программы с инструкциями `goto`, которые писались на заре программирования. Но это скорее проблема начинающих программистов, чем языка программирования. Сдержанность и чёткая документация отличают профессионалов от новичков.

Контрольные вопросы

1. О чем свидетельствует одинарный символ подчёркивания в начале имени переменной?
 - а) Эта переменная видна только для Python.
 - б) Созданная внутри класса, эта переменная не будет видна другим классам.
 - в) Эта переменная не импортируется инструкцией `from <module> import *`.
 - г) Python ограничивает доступ к этой переменной.
2. Что означает слово `self`?
 - а) Иллюзия, состоящая из пяти компонентов.
 - б) Так же, как и указатель `this` в C++, указывает на текущий объект.
 - в) Метод изменения объектом самого себя.
 - г) Ключевое слово, используемое для создания полиморфизма.
3. Какая ошибка множественного наследования может привести к неправильному выполнению методов смешанного класса?
 - а) Наличие одноимённых методов или переменных в родительских классах.
 - б) Использование в родительском классе методов, выполняющих тригонометрические вычисления.
 - в) Использование в родительском классе рекурсивных методов.
 - г) Неправильный порядок следования в списке родительских классов, в результате чего активным становится не тот метод, который ожидался.

Ответы

1. в. Переменные, чьи имена начинаются с одного символа подчёркивания, не импортируются в модуль инструкцией `from <module> import *`.
2. б. В Python `self` означает то же, что в C++ указатель `this`. С помощью `self` объект может изменять сам себя, но это не готовый метод. Будда говорит, что *self* – это иллюзия, но это не верно для Python.
3. г. Последовательность, в которой родительские классы представлены в списке, определяет выбор активного метода или переменной, если в другом родительском классе содержатся одноимённые члены. Одноименные члены классов не только возможны, но и необходимы для поддержания полиморфизма объектов.

Примеры и задания

Создайте смешанный класс, описанный в начале раздела "Множественное наследование", который бы предоставлял метод `__repr__()` всем классам, унаследованным от него. Затем создайте смешанный класс на основе созданного класса и класса `today`.

Чтобы класс `today` продолжал работать, в нём нужно изменить имя метода `__repr__()` на какое-нибудь другое.

Попробуйте создать программу, работающую из командной строки, которая бы распознавала символы римских цифр и преобразовывала их в десятичные значения.

Попробуйте сократить код класса `roman`, используя для литералов списки или массивы.

Чтобы больше узнать о концепции личности в буддизме, обратитесь к статье Будхадаза Бхиккху (Buddhadasa Bhikkhu) *The Buddha's Teaching on Voidness* (Учение Будды о пустоте), представленную на сервере Wisdom Publications (<http://www.channell.com/users/wisdom/index.html>).

13-й час

Специальные методы классов в Python

В этой главе мы попытаемся свести вместе известные Вам факты. Мы рассмотрим специальные методы классов – эти забавные функции, имена которых начинаются и заканчиваются двумя символами подчёркивания. К концу главы Вы научитесь добавлять собственные специальные методы в написанные Вами классы.

Операторы и перегрузка операторов

В главе 3 Вы узнали об операторах и операндах. Вы также узнали, что операторы, например суммирования (+), могут вести себя по-разному в зависимости от типа подставленных операндов. Например, выражения `1+1` и `"Hello, "+"World!"` будут выполняться совершенно по-разному. Если в качестве операндов используются числа, оператор + складывает их, если строки – то выполняет конкатенацию. Это типичный пример полиморфизма – два разных ответа на одно и то же сообщение "сложить", переданное разным объектам. В первом случае сообщение передаётся двум числовым объектам, во втором – двум строковым. Полиморфизм в данном случае был достигнут за счёт процедуры, называемой *перегрузкой операторов*. (Чтобы быть более точным, речь идёт о перегрузке имени оператора, т.е. допускается существование в одном пространстве имён нескольких функций с одинаковыми именами, но с разными наборами параметров и выполнениями.) Под этим термином понимается определение различного поведения операторов в зависимости от контекста. Поведение встроенных операторов при обработке базовых типов данных не может быть изменено в Python, но Вы вольны самостоятельно определять работу того или иного оператора при использовании созданных Вами объектов. Причем не только в тех случаях, когда оба операнда относятся к пользовательским типам данных, но и в случае выполнения операций над объектом и переменной базового типа. Перегрузка операторов, для выполнения которой используются специальные методы классов Python, является одним из самых мощных и эффективных средств программирования. В других современных языках также допускается перегрузка операторов, но нигде это не делается так просто, как в Python.

***Прим. В. Шипкова: эта методика была явно позаимствована из других языков, в частности C++. ИМХО, Гвидо превзошёл Бьёрна! ;)))**

Очень полезно иметь возможность определять поведение объектов в своей программе в ответ на сообщения разных типов. В языке C нельзя перегружать операторы.

Существует возможность только создавать пользовательские функции для выполнения специфических задач, но вызов функции в C всегда осуществляется только по имени без учёта типов операндов. Так, если Вам в программе нужно особым образом обрабатывать пару объектов, например сложить их, то следует написать специальную функцию, куда эти объекты передаются с аргументами. Такой подход усложняет код и делает его трудным для чтения и понимания. Например, когда

я только увлёкся календарем индейцев Майя, для своей программы мне пришлось создать целую библиотеку функций, позволяющих выполнять операции с целыми числами произвольной длины. Я устал от необходимости постоянно контролировать тип переменной, так как в своих вычислениях мне приходилось оперировать цифрами, лежащими на границе допустимых значений для типа `int`. Чтобы решить проблему, мне пришлось разработать набор функций вроде тех, что используются в Python для обработки длинных целых значений. Поскольку моя библиотека и программы были написаны на языке C, приходилось явно вызывать каждую функцию, так как перегрузка операторов была недоступна. У меня возникла необходимость умножить переменные `ram` на `tt`, а затем разделить результат по модулю на 13. Но вместо простого однострочного выражения пришлось вводить целый блок определений и вызовов функций:

```
Hint * tmpn = mpIntToMint(0);
Mint * q = mpIntToMint(0);
Mint * r = mpIntToMint(0);
Mint * tx = mp!ntToMint(13);
Mint * mm = mp!ntToMint(20);
Mint * tt = mp!ntToMint(819);
mpMultiply(mm, tt, tmpn); /* Вывод заносится в tmpn */
mpDivide(tmpn, tx, q, r); /* Деление по модулю на 13,
результат в r */
mpPtrFree (mm, modname) ; /* Очистка временных переменных
*/
mpPtrFree(tt, modname); mpPtrFree(q, modname);
mpPtrFree(tmpn, modname); mpPtrFree(tx, modname);
return (r);
```

Позже, когда я стал использовать C++, я захотел переписать свою библиотеку с использованием появившейся возможности перегрузки операторов. Но, к счастью, в этот момент я узнал о Python и о том, что в нём весь этот блок можно просто заменить одним выражением:

```
r = (20L * 819L) % 13
return r
```

или даже

```
return (20L * 819L) % 13L
```

Но и это не всё. Оказалось, что в Python я могу очень просто создать свой пользовательский класс, который избавит меня от проблем подобного рода. Достаточно будет просто импортировать мой класс, как в следующем примере:

```
from mayalib import *
x = mayanum()
x()
x = x + "13.0.0.0.0"
x()
print x.gregorian()
```

В языке С для выполнения этой же задачи пришлось бы написать множество строк кода, затем ещё потратить время на их отладку, а программа в результате получилась бы громоздкой и трудночитаемой. (Эта программа, между прочим, рассчитывает дату конца света в соответствии с верованиями индейцев Майя и преобразовывает её в дату григорианского календаря. Вы можете загрузить эту программу с Web-страницы по адресу [http://www.pauahtun.org/TYPython/.](http://www.pauahtun.org/TYPython/))

Часто вполне очевидно, что данный класс должен делать в ответ на определённое сообщение или применение оператора. Например, в случае с датами календаря Майя смысл применения оператора + вполне понятен. Он используется для суммирования одной даты в формате календаря Майя с другой датой в этом же формате или другом, понятном для функции оператора. Но не всегда все так просто. В главе 15 мы займёмся разработкой программы для автоматизации записи на видеомagnetofон выбранных телепередач. В этой программе не так-то просто будет разобраться, что именно выполняет оператор + с пользовательскими объектами, когда суммирует телепередачу с видеокассетой. Но если задуматься, как происходит запись передачи в реальной жизни, то сами собой появятся идеи, что должен делать оператор суммирования, хотя выполнение этих задач функцией оператора может быть реализовано по-разному. Давайте попробуем найти одно из логических решений. Создание программы всегда следует начинать с анализа логики её выполнения, что существенно сократит число проб и ошибок во время написания кода программы.

Магнитная лента — это контейнер. Контейнер может быть пустым, может содержать одну или несколько записанных программ, а может даже случиться так, что в кассете уже не останется места для записи новой программы. Иногда магнитофон может "зажевать" ленту, и тогда нужно прервать запись и звонить мастеру. Таким образом, нам потребуется класс cassette (кассета), который может быть пустым, а может содержать экземпляры класса program. Что мы делаем с обычной видеокассетой? Мы записываем на неё передачу, а когда она надоест, можем стереть её. Значит, нам нужен класс, который знает, что делать с объектами других классов, т.е. может добавлять их в состав своего объекта и

обработать по командам пользователя. Пришло время заняться планированием класса — важнейшим этапом объектно-ориентированного программирования на любом языке. Конструирование объекта с определёнными свойствами, содержащего другие объекты, свойства которых, в свою очередь, являются предметом такого же анализа, — это фундамент программирования классов. Но если в других языках под планированием понимают чисто умозрительный процесс с вычерчиванием блок-схем на листке бумаги, то в Python, учитывая предельную простоту программирования, планирование удобно сочетать с экспериментированием разных вариантов выполнения класса, сидя прямо у экрана компьютера. Такой подход позволяет сразу оценить на практике, насколько точно в данном варианте класса реализованы ваши идеи. Представления о том, каким класс должен быть, проще всего выкристаллизовать путём многократного изменения и настройки его функциональности. Такой диалог между программистом и машиной может быть чрезвычайно продуктивен, главное — не бояться пробовать и экспериментировать. Отсутствие необходимости в Python перекомпилировать и связывать программные модули после внесения каждого изменения даёт возможность программисту вволю поупражняться с кодом программы, не рискуя при этом потерять много времени и выпасть из графика сдачи проекта.

Мы ещё вернёмся к классам кассеты и программ чуть позже, в главе 16. Но чтобы в полной мере раскрыть эту тему, сейчас нам нужно ближе познакомиться со специальными методами классов.

Использование специальных методов

В приложении В Вы найдёте отсортированный в алфавитном порядке полный список всех специальных методов классов, представленных в Python 1.5.2. Все эти методы могут быть использованы в пользовательских классах. Если у программиста возникнет такая необходимость, он может разработать особое выполнение данного метода, чтобы обогатить им функциональность своего класса. Обратите внимание, что к этому средству следует прибегать только в случае необходимости. Стремление молодых программистов переписать все специальные методы на свой лад часто выливается в пустую трату времени. Кроме того, новые функциональные возможности класса должны быть логичными. Например, в предыдущем разделе мы показали, что для класса `cassette` было бы логично иметь перегруженный оператор суммирования (+) для добавления в контейнер новых записей программ (объектов класса `program`). Но какой смысл в этом классе может иметь оператор умножения (*)? Вы вольны

приписать этому оператору любую функцию, например очистку контейнера от всех записей программ. Программа будет работать, но вашим пользователям будет непонятно, каким образом оператор умножения связан с удалением записей, что сделает интерфейс вашей программы недружественным для пользователя. Видимо, в классе `cassette` следует ограничиться перегрузкой операторов `+` и `-`, поскольку их применение в большей степени соответствует представлениям пользователей о происходящих процессах.

Большинство специальных методов классов требуют передачи им определённых аргументов и в определённом порядке, а в ответ они возвращают значения определённого типа. Некоторые другие методы значительно более гибкие в применении, хотя общим требованием для всех специальных методов будет передача аргумента `self` первым в списке параметрам. Как Вы помните, `self` — это простой и явный метод указания методу текущего типа вызывающего объекта, на основании чего происходит выбор корректной версии метода для этого типа данных. Вы уже встречались с примерами реализации полиморфизма специальных методов, например при вызове метода `__init__()`. Единственный аргумент, требуемый этим методом, является `self`, а сам метод не возвращает никаких значений. Впрочем, при написании кода класса Вы можете потребовать передачу дополнительных аргументов методу `__init__()` для создания экземпляра, своего класса. При запуске программы Python проследит за тем, чтобы пользователь ввёл все аргументы, которые Вы сделали обязательными для своего класса, хотя встроенному методу `__init__()` достаточно одного аргумента `self`.

Это означает, что Вы можете оставить метод `__init__()` таким, какой он есть. В этом случае он ничего не будет делать в Вашем классе, а для настройки состояния и функциональности класса можно будет разработать свой методы. Чтобы метод `__init__()` ничего не делал в Вашем классе, достаточно написать следующее:

```
class egg:
def __init__(self):
    pass
```

Инструкция `pass` в определении класса несёт тот же смысл, что и при её употреблении в конструкциях с инструкциями `if` или `try-except`, т.е. она просто сообщает Python, что лучше в этом месте ничего не делать.

Другие специальные методы классов, особенно те из них, что используются для обслуживания операторов действий с

числовыми значениями, гораздо более требовательны к числу и типу аргументов. Для примера рассмотрим метод `__add__()`, который может принимать только два аргумента, `self` и `other`, и возвращает новый объект, который представляет собой результат суммирования или конкатенации значений `self` и `other`. Впрочем, Вы можете определить для этого метода выполнение какой-либо другой операции над этими двумя переданными объектами. Желательно только, чтобы эта операция логически как-то соотносилась с процессом суммирования. Например, не разумно заставлять метод `__add__()` выполнять умножение двух чисел! Но для нашего предыдущего примера с добавлением объекта телепередачи к объекту кассеты этот метод вполне подойдёт, хотя реальное суммирование значений здесь не происходит.

В качестве примера приведём ещё одну область программирования, где перегрузка операторов будет весьма полезной. Так, в теории множеств используются понятия соединения, объединения и пересечения, для которых в математике существуют специальные операторы. Но эти операторы, к сожалению, отсутствуют в Python. В таком случае для Вас будет вполне оправдана перегрузка операторов `+` — для выполнения соединения множеств, `|` — для объединения и `&` — для пересечения. Любой, кто хоть немного разбирается в теории множеств, быстро запомнит смысл использования операторов в вашей программе.

Метод `__coerce__()` в месте с другими специальными методами операторов, такими как `__add__()`, будет весьма эффективен для решения задач, связанных с преобразованием объектов разных типов, как базовых, так и пользовательских, в экземпляр Вашего класса. У этого метода также достаточно жёсткие требования к аргументам и возвращаемому значению. Он принимает два аргумента, `self` и `other`, а возвращает их же в виде набора (`self`, `other`). В качестве аргумента `other` может выступать значение базового типа, экземпляр того же класса, что и `self`, или объект любого другого типа. Только нужно продумать, какие типы данных будут допустимы для Вашего конкретного класса, и разработать соответствующие выполнения метода `__coerce__()` и структуру возвращаемого им набора для разных типов. В возвращённом наборе элемент `other` должен быть представлен новым экземпляром класса, заданного аргументом `self`. (Хотя в принципе допустимо приводить с помощью этого метода аргумент `self` к типу `other`, такое решение будет выглядеть довольно странно. Нужно иметь достаточно веское основание, чтобы так поступить, я даже не могу придумать, какое именно.)

Прежде чем перейти к детальному рассмотрению применения на практике наиболее важных специальных методов, давайте остановимся на особенностях ещё одного встроенного метода – `__del__()`. Во многих объектно-ориентированных языках программирования поддерживается техника *сбора мусора*, под которой понимают способность интерпретатора время от времени просматривать все когда-либо созданные объекты и избавляться от тех из них, которые больше не используются. Python также имеет встроенный механизм избавления от мусора, но в основе его лежит несколько иной принцип, называемый *счётчиком ссылок*. Он состоит в том, что каждый раз при создании объекта Python добавляет в него скрытый атрибут, представляющий собой счётчик числа функций, классов и методов, использующих этот объект. Если никто больше не использует этот объект в программе, Python удаляет его из памяти компьютера, предоставляя освободившиеся ячейки памяти другим объектам. В следующем примере показано определение функции с локальной переменной `i`:

```
def spam(y):  
    i = y * 400  
    ... другие выражения ...
```

При создании переменной `i` к небольшому объекту, представляющему данную переменную, добавляется скрытый счётчик с присвоенным значением 1. Это число указывает, сколько пользователей в данный момент используют переменную `i`. По завершении выполнения функции, когда переменная `i` выходит из поля зрения, очевидно, что никто больше не будет использовать эту переменную, поэтому интерпретатор отнимает от значения счётчика единицу. При этом он обнаруживает, что значение счётчика стало равным 0, поэтому этот объект удаляется. Хотя существует много других алгоритмов сбора мусора, вариант со счётчиком ссылок отличается стабильностью, ошибкоустойчивостью и простотой. С точки зрения Guido, наиболее важной особенностью данного алгоритма является его переносимость, тогда как другие более сложные алгоритмы сбора мусора, как правило, зависимы от платформы и системы компьютера. Кроме того, многие алгоритмы периодически прерывают работу интерпретатора на время их выполнения, чего не происходит с алгоритмом счётчика ссылок. Такие задержки в ответе программы на команды могут раздражать пользователя или даже восприниматься как ошибки программы. Недостатком использования алгоритма счётчика ссылок в таком динамичном языке программирования, как Python, является то, что не всегда удаётся добиться надёжности в определении текущего состояния счётчика. (Тут много причин, почему могут

происходить сбои, но эта тема выходит за рамки данной книги.) В случае возникновения неопределённости Python предпочитает не удалять подозрительные объекты. Сохраняя объект, Python рискует, что после завершения программы в памяти компьютера могут остаться заблокированные ничейные ячейки памяти, к которым никто уже не сможет обратиться. Впрочем, ничего страшного с компьютером не происходит, если не считать того, что часть памяти временно становится недоступной для использования другими программами. Для большинства современных компьютеров небольшая утечка памяти не будет такой уж серьёзной проблемой.

Метод `__del__()` автоматически вызывается Python для удаляемых объектов. Существует также встроенная функция `del()`, но в действительности она сама не удаляет объект, а уменьшает на единицу значение его счётчика ссылок. Если после этого значение счётчика становится равным 0, то остальное делает метод `__del__()`. Пример удаления объекта показан в листинге 13.1.

Листинг 13.1. Удаление объекта, программа del.py

```
#!c:\python\python.exe

class spam:
    def __init__(self):
        pass
    def __del__(self):
        print "I'm about to be deleted!"

a = spam()

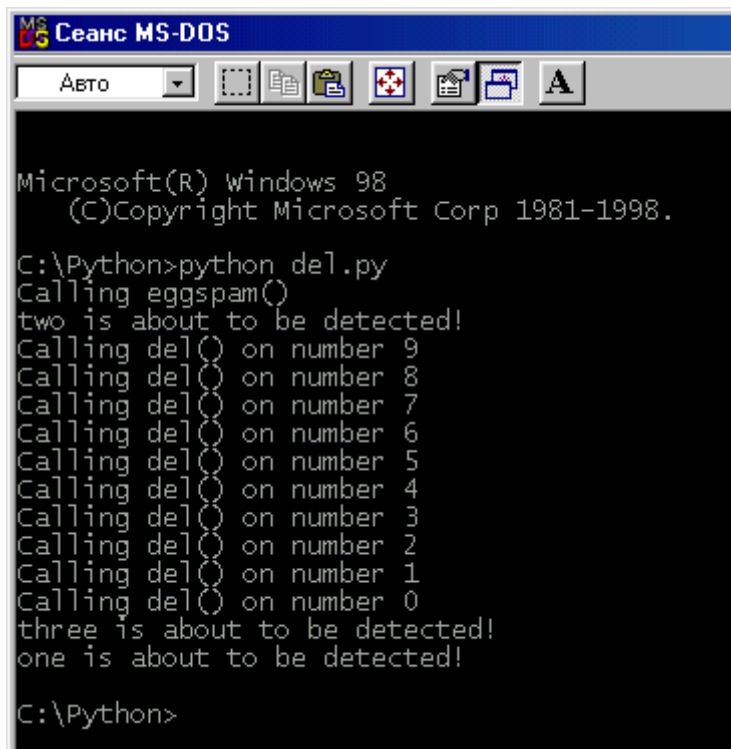
def eggspam():
    z = spam()

if __name__ == "__main__":
    print "Calling eggspam()"
    b = eggspam()
    t = []
    for i in range(10):
        t.append(a)

    for i in range(9,-1,-1):
        print "Calling del() on number", i
        del(t[i])

    x = spam()
```

Результат выполнения программы del.py показан на рис. 13.1.



```
Microsoft(R) Windows 98
(C)Copyright Microsoft Corp 1981-1998.

C:\Python>python del.py
Calling eggspam()
two is about to be detected!
Calling del() on number 9
Calling del() on number 8
Calling del() on number 7
Calling del() on number 6
Calling del() on number 5
Calling del() on number 4
Calling del() on number 3
Calling del() on number 2
Calling del() on number 1
Calling del() on number 0
three is about to be detected!
one is about to be detected!

C:\Python>
```

Рис. 13.1. Результат выполнения программы del.py

Функция `eggspam()` создаёт объект `z` класса `spam`, который тут же выходит из поля видимости с завершением функции. Список `t` содержит 10 ссылок на объект `a` класса `spam`. По мере удаления элементов списка `t` в обратном порядке в действительности происходит только уменьшение значения счётчика ссылок объекта `a`. И только когда значение счётчика становится равным нулю, действительно удаляется объект. Наконец, мы создаём ещё один объект `x` класса `spam`, который явно в программе не удаляется. Но Python при завершении программы автоматически удаляет все созданные в ней объекты, сообщение о чём и появилось на экране.

Всегда, когда Вам нужно удалить все элементы списка, делайте это в обратном порядке, как это происходит во втором цикле `for` в листинге 13.1. Дело в том, что при удалении элемента списка автоматически уменьшается его размер. Так, в нашем примере при удалении элементов списка от 0-го до 9-го в момент удаления 5-го элемента в списке не существовало элементов с индексом больше 5. Чтобы убедиться в этом, измените второй цикл `for` следующим образом:

```
for i in range(10):
```

Теперь вновь запустите программу `del.py` и внимательно прочтите появившееся сообщение об ошибке: `IndexError: list assignment index out of range` (индекс указывает за пределы списка).

Числовые специальные методы классов

В этом разделе мы используем уже знакомый Вам класс `spam` (фарш) и на его примере продемонстрируем использование основных числовых специальных методов классов. Специальные методы классов разделены на группы по тому признаку, на работу с какими типами данных они ориентированы. Выделяют числовые методы, методы для работы с пользовательскими классами, методы для обработки последовательностей и методы доступа. Первый из числовых специальных методов классов, к которому мы обратимся, – это метод `__add__()`.

Как Вы помните, метод `__add__()` требует наличия двух аргументов, `self` и `other`, а должен вернуть объект типа `self`. (Это не совсем верно говорить, что метод `__add__()` обязательно должен возвращать объекты типа `self`. В действительности допускается прямо в теле метода перед возвращением изменить тип объекта на другой, но это не считается хорошей практикой программирования. Давайте снабдим наш класс `spam` дополнительной функциональностью, как показано в листинге 13.2.

Листинг 13.2. Выполнение метода `add()` в программе `spam.py`

```
#!/c:\python\python.exe
class spam:
    def __init__(self):
        self.eggs = 1

    def __del__(self):
        pass

    def __add__(self, other):
        rt= spam()
        rtreggs = self.eggs + other.eggs
        return rt

if __name__ == "__main__":
    a = spam()
    b = spam()
    a = a + b
    print "a now has", a.eggs, "eggs"
```

Если Вы сейчас запустите `spam.py`, то программа сообщит Вам: `a now has 2 eggs` (в фарш 'a' добавлено 2 яйца). Таким образом, с помощью метода `__add__()` мы сложили наши виртуальные яйца. Точно так же мы можем определить метод

`__sub__()`, только вместо сложения применить вычитание. Также не должно возникнуть проблем с определением для класса `spam` методов `__mul__()` и `__div__()`, которые соответственно будут умножать и делить объекты. Определения большинства числовых методов вполне очевидны. Полный список специальных числовых методов классов Вы найдёте в приложении Г.

Теперь предположим, что Вам нужно выполнить сложение объекта `spam` и строки символов, например следующие выражения:

```
a=spam()
a=a+"24"
a ="24"+a
print a.eggs
```

Чтобы снабдить класс такой функциональностью, нужно добавить выполнение ещё двух специальных методов: `__coerce__()` и `__radd__()`. Для упрощения кода мы будем добавлять типы в метод `__coerce__()`, только по мере возникновения потребности в них. Так что на этом этапе добавим только функцию обработки строковых переменных, как показано в листинге 13.3.

Листинг 13.3. Добавим в фарш ещё яиц

```
#!c:\python\python.exe
import string

class spam:
    def __init__(self):
        self.eggs = 1

    def __del__(self):
        pass

    def __add__(self, other):
        rt = spamf)
        rt.eggs = self.eggs + other.eggs
        return rt

    def __coerce__(self, other):
        rt = spam()
        if type(other) == type(rt):
            return (self, other)
        elif type(other) == type(""):
            e = string.atoi(other)
```

```

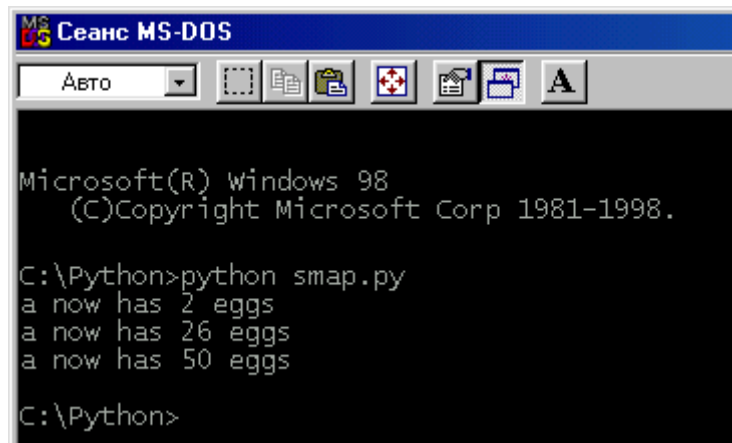
        rt.eggs = e
        return (self,rt)
    else:
        return None

    def __radd__(self,other):
        return self+other

if __name__ == "__main__":
    a=spam()
    b=spam()
    a=a+b
    print "a now has", a.eggs, "eggs"
    a=a+24
    print "a now has", a.eggs, "eggs"
    a=a+24
    print "a now has", a.eggs, "eggs"

```

На рис. 13.2 показан результат выполнения измененной программы spam.py.



```

Microsoft(R) Windows 98
(C)Copyright Microsoft Corp 1981-1998.

C:\Python>python spam.py
a now has 2 eggs
a now has 26 eggs
a now has 50 eggs

C:\Python>

```

Рис. 13.2. Фарш и яйца

Обратите внимание, что в методе `__radd__()` нам не пришлось делать ничего особенного. Python лишь требует, чтобы последовательность аргументов в этом методе сохранялась такой же, как и в методе `__add__()`. Поскольку у нас уже есть в программе выполнение метода `__add__()`, то метод `__radd__()` просто вызывает его из своего тела.

Примите это как общее правило. Если в определении класса указано выполнение нормального числового метода и метода `__coerce__()`, то любой вспомогательный метод `__r*__()` выполняет вызов соответствующего нормального метода, благодаря чему Вам не нужно переписывать код дважды. Это лишь один из примеров встроенной возможности Python принимать правильные решения без лишних указаний.

Поскольку класс `spam` оперирует числовыми значениями, можно добавить в него дополнительные специальные числовые методы классов, такие как `__abs__()`. Полный вариант определения класса `spam`, содержащего выполнение большинства специальных числовых методов классов, показан в листинге 13.4.

Листинг 13.4. Специальные числовые методы классов

```
#!/c:\python\python.exe
import string

class spam:
    def __init__(self):
        self.eggs = 1

    def __Jel__(self):
        pass

    def __add__(self, other):
        rt = spam()
        rt.eggs = self.eggs + other.eggs
        return rt

    def __coerce__(self, other):
        rt = spam()
        if type(other) == type(rt):
            return (self, other)
        elif type(other) == type(""):
            e = string.atoi(other)
            rt.eggs = e
            return (self, rt)
        elif type(other) == type(0):
            rt.eggs = other
            return (self, rt)
        else:
            return None

    def __radd__(self, other):
        return self + other

    def __absj__(self):
        return abs(self.eggs)

    def __long__(self):
        return long(self.eggs)

    def __float__(self):
        return float(self.eggs)
```



```

def __int__(self):
    return int(self.eggs)

def __complex__(self):
    return complex(self.eggs)

def __divmod__(self, other):
    return divmod(self.eggs, other.eggs)

def __and__(self, other):
    return self.eggs & other.eggs 42:

if __name__ == "__main__":
    a = spam()
    b = spam()
    a = a + b
    print "a now has", a.eggs, "eggs"
    a = a + "24"
    print "a now has", a.eggs, "eggs"
    a = "24" + a
    print "a now has", a.eggs, "eggs"
    b = b + "13"
    print divmod(a,b)
    print a & b

```

В качестве домашнего задания найдите и добавьте в класс `spam` выполнение оставшихся специальных числовых методов классов. Это не очень сложное задание.

Специальные методы классов для обработки последовательностей и установки отношений

В этом разделе мы создадим образец класса изменяемых строк, применение которых иногда бывает весьма эффективным. В листинге 13.5 показан первый вариант класса `parrots` (попугаи).

Листинг 13.5. Класс `parrots`

```

#!c: \python\python . exe
import string
import sys

class parrot:
    def __init__(self,s=""):
        self.s = list(s)

```

```

def repr (self):
    t = ""
    t=string. join(self . s, ' ')
    return t

def __str__(self):
    return parrot, repr (self)

def __add__( self, other):
    rt = parrot( 'self ')
    rt.s = rt.s + other. s
    return rt

def radd (self , other) :
    return self + other

def coerce (self , other):
    if type (other) == type (self):
        return (self, other)
    elif type (other) == type(""):
        rt = par rot (other)
        return (self, rt)
    elif type (other) == type([]):
        print "type list"
        rt = parrot( )
        for i in other:
            if len(i) > 1:
                for j in i:
                    rt.s.append(j)
            else:
                rt.s.append(i)
        return (self, rt)
    elif type(other) == type({}):
        return None
    else:
        rt = parrot(str(other))
        return (self, rt)

def __getitem__(self, key):
    if type(key) == type(0):
        return self.s[key]
    else:
        raise TypeError

def __getslice__(self,i,j):
    s = self.s[i:j]
    t = ""
    for k in s:

```

```

        t = t + k
    return t

    def __setitem__(self, key, value):
        if type(key) == type(0):
            self.s[key] = value
        else:
            raise TypeError

    def __setslice__(self, i, j, value):
        self.s[i:j] = list(value)

    def __len__(self):
        return len(self.s)

if __name__ == "__main__":
    if len(sys.argv) > 1:
        s = sys.argv[1]
    else :
        s = "I'm a little teapot!"
    p = parrot(s)
    print p
    q = parrot (" I'm not schizophrenic!")
    print "Add:", p + q
    x = ['N', 'o', ", that bird's not dead", "!"]
    print "Add:", p + ' ' + x
    print "Index[9]: ", p[9]
    print "Index[-9]: ", p[-9]
    p[11], p[13] = p[13], p[11]
    p[14], p.[16] = p[16], p[14]
    print p
    pa = p[11:17]
    print pa
    print "Add:", p + ' ' + q
    p[11:17] = "parrot"
    print p, len(p)

```

В классе parrot используется лишь несколько специальных методов классов. При желании можно было бы добавить ещё много дополнительных методов. Например, полезным может оказаться выполнение метода `__str__()`, с помощью которого можно сортировать объекты в списке.

Особое внимание обратите на приём, использованный в строке 76: `p[11], p[13]=p[13], p[11]`. Аналогичное выражение содержит строка 77. Такой подход можно использовать каждый раз, когда нужно выполнить обмен значениями между двумя переменными. В других языках для этого потребовалось бы

создание временной переменной для сохранения промежуточных значений, как в следующем примере:

***Прим. В. Шипкова: приведённый пример, ИМХО, весьма стильное решение.**

```
tmp = a;  
a = b;  
b = tmp;
```

Но в Python эту операцию можно выполнить в одной строке:

```
a, b = b, a
```

Если добавить ещё специальные методы, у Вас получится великолепный класс с изменяемыми строковыми значениями, который можно использовать в самых разных программах. В следующей главе Вы подробнее познакомитесь с техникой многократного использования в своих программах стандартных блоков, таких как класс `parrot`.

Специальные методы классов и вопросы управления доступом

Ещё один аспект использования специальных методов классов, который осталось рассмотреть, — это управление с их помощью доступом к членам класса. Методы, которые используются для того, чтобы пользователь мог возвращать или устанавливать значения переменных-членов класса, называются *методами доступа*. К ним можно отнести такие специальные методы, как `__call__()`, `__del__()`, и некоторые другие. Методы доступа определяют способ использования объекта в программе как единого целого. Эти методы также важны для организации взаимоотношений между объектами. Например, метод `__cmp__()` используется при сравнении в выражениях, подобных следующему:

```
if a < b:
```

Назначение большинства специальных методов классов самоочевидно, за исключением разве что метода `__init__()`, который мы уже рассматривали, и метода `__hash__()`, применяемого в тех случаях, когда неизменяемый объект используется в качестве ключа словаря. Этот метод используется довольно редко, поскольку в программах чаще нужны изменяемые объекты. Но если у Вас возникнет необходимость в таких словарях и неизменяемых объектах, информацию о них, конечно же, можно найти на уже хорошо

известной Вам домашней странице Python
(<http://www.python.org/>).

Для изменяемых объектов правильное использование методов доступа может оказаться крайне важным. С их помощью можно контролировать взаимодействие пользователей класса с его членами, что позволяет предупредить возможное повреждение класса пользователем или неправильное его использования.

***Прим. В. Шипкова: это замечание более чем справедливо. Действия профессионала предсказуемы. Но в этом мире полно любителей.**

К важным специальным методам доступа можно отнести `__delattr__()`, `__getattr__()` и `__setattr__()`.

Первый метод используется при удалении атрибута класса с помощью функции `del()`. Чтобы предупредить удаление, в методе `__delattr__()` можно установить вызов какого-нибудь подходящего исключения с предварительным выводом сообщения для пользователя. Если удаление атрибута допускается, то эта процедура должна выполняться на уровне словаря объектов. Другими словами, метод `__delattr__()` нельзя выполнять подобным образом:

```
def __delattr__(self, имя):  
    ...определение удаляемых атрибутов...  
    del(self.имя)
```

От такого выполнения ваша программа сойдет с ума, так как метод `__delattr__()` автоматически выполняется каждый раз при вызове функции `del()`, в результате чего получится бесконечный цикл. Правильно будет выполнить удаление так:

```
def __delattr__(self, имя):  
    ...определение удаляемых атрибутов...  
    del (self.__dict__[имя])
```

В результате будете удовлетворены и Вы, и Python.

Для ситуации, описанной выше, в программировании применяется специальный термин – рекурсия. Под рекурсией понимают вызов функцией самой себя. Бесконечная рекурсия возникает тогда, когда функция вызывает себя снова и снова и нет никакого способа остановить этот процесс. С другой стороны, рекурсивные функции могут оказаться весьма полезными, но обязательно нужно предусмотреть способ их остановки. В нашем примере с методом `__delattr__()` и

функцией `del()` проблема возникнет именно из-за того, что это бесконечная рекурсия.

Второй метод `__getattr__()` вызывается каждый раз, когда пользователь пытается возвратить несуществующий атрибут объекта. Например, в рассмотренном выше классе `parrot` представлен единственный атрибут состояния – список `s`. С помощью метода `__getattr__()` можно, например, определить возвращение строки списка `s` каждый раз, когда пользователь пытается возвратить любой другой атрибут:

```
def __getattr__(self, имя):  
    return str(self.__dict__["s"])
```

Наконец, третий метод, `__setattr__()`, вызывается при попытке пользователя добавить в класс новый атрибут, т. е. создать новую переменную-член класса. Как и в предыдущих двух случаях, эта процедура доступа к атрибутам должна выполняться на уровне словаря класса. Но выполнение этого метода может оказаться более сложной задачей. Не забудьте также, что этот метод вызывается при создании всех атрибутов, даже тех, чьё создание было предопределено в коде класса. В листинге 13.6 показаны примеры использования всех трёх методов доступа. Особенно внимательно проанализируйте выполнение метода `__setattr__()`.

Листинг 13.6. Класс `egg`

```
#!c:\python\python.exe  
import string  
import sys  
  
class egg:  
    def __init__(self):  
        self.yolks = 1  
        self.white = 1  
        self.brains = 0  
        self.spam = 0  
  
    def __delattr__(self, name):  
        if name == "spam":  
            print "All eggs .mast have spam!"  
            raise AttributeError  
        print "deleting", name  
        del(self.__dict__[name])  
  
    def __setattr__(self, name, value):  
        try:
```

```

        s = self.__diet__[name]
    except KeyError:
        if name not in [" yolks", "white", "brains", \
            "spam", "salt", "pepper"]:
            print "you're not allowed to add", \
                name, "to eggs"
        else:
            s = self.__diet__[name] = value
    else:
        s = self.__diet__[name] = value

def __getattr__(self, name):
    if name != " yolks":
        return self.__diet__["spam"]
    else:
        raise AttributeError

x = egg()
print x.yolks
x.spam = 1
X.snot = 1
try:
    del x.spam
except AttributeError:
    print "I couldn't get rid of the spam!"
    print x.spam
    x.spam = 1000
    print x.spam
    del x.yolks
try:
    print x.yolks
except AttributeError:
    print "No Yolks!"
    x.white = 29
    print "Snot is", x.snot, "white is", x.white

```

Обратите внимание, как для упрощения кода в выполнении метода `__setattr__()` используется конструкция `try...except...else` (строки 17–26).

Резюме

В этой главе Вы узнали о специальных методах классов. Теперь Вы умеете использовать специальные методы классов для перегрузки операторов; выполнять инициализацию и настройку состояния объекта класса; распознавать различные категории специальных методов классов и разбираться в особенностях их использования; управлять добавлением, возвращением и удалением атрибутов класса.

В следующей главе Вы познакомитесь с многократно используемыми компонентами программ: как их использовать, создавать и находить, чтобы не создавать самостоятельно.

Практикум

Вопросы и ответы

Зачем нам использовать инструкцию `pass`?

Возможно, было бы гораздо проще просто оставлять пустыми те строки в методах классов, циклах и конструкциях с инструкциями `if`, где программа не должна выполнять никаких действий. Но использование инструкции `pass` позволяет явно указывать необходимость ничего не делать в этом месте программы. По крайней мере Вы будете уверены, что другой программист, работающий с вашей программой, или Вы сами по прошествии времени не решите, что код в данном месте был пропущен по ошибке.

Можно ли создать в программе классы, которые ссылаются друг на друга? Например, сделать так, чтобы класс `spam` включал объекты класса `egg`, а класс `egg` мог содержать объекты `spam`.

Возможно, но в этом случае возникают *циклические ссылки*, а Python не умеет корректно поддерживать их работу. В результате может возникнуть существенная утечка памяти. Тем не менее в некоторых случаях создания циклических ссылок просто невозможно избежать.

Контрольные вопросы

1. Назовите четыре категории специальных методов классов.

- а) Числовые, строчные, для обработки пользовательских классов и доступа.
- б) Числовые, для обработки последовательностей, для работы с пользовательскими классами и методы доступа.
- в) Гомерические, ионические, дорические и коринфские.
- г) Существенные, полусущественные, несущественные, абстрактные.

2. Выберите три специальных метода доступа.

- а) `__del__()`, `__init__()` и `__cmp__()`.
- б) `__add__()`, `__radd__()` и `__mul__()`.
- в) `__delattr__()`, `__getattr__()` и `__setattr__()`.
- г) `__and__()`, `__or__()` и `__xor__()`.

3. Каким способом в Python проще всего выполнить обмен значениями двух переменных?

- а) `tmp = a, a = b, b = tmp`
- б) `a ^ b`
- в) `a, b = b, a`
- г) `swap(a, b)`

Ответы

1. б. Выделяют числовые специальные методы, методы для обработки последовательностей, методы для работы с пользовательскими классами и методы доступа. Вполне возможно, что в следующих версиях Python появятся новые категории специальных методов.
2. в. К специальным методам доступа относятся `__delattr__()`, `__getattr__()` и `__setattr__()`.
3. в. В Python проще всего произвести обмен значениями между двумя переменными с помощью выражения `a, b = b, a`, так как в этом случае удаётся избежать создания временных переменных.

Примеры и задания

Добавьте выполнение оставшихся числовых методов в класс `spam`, представленный в листинге 13.4.

Добавьте специальные методы в класс `parrot` из листинга 13.5. Хотя в классе уже представлены два метода обработки последовательностей, можно ещё добавить, например, метод `__sub__()`, который будет просматривать строку `self` и отнимать от неё строку `other`, если таковая будет найдена. Полезным в этом классе может также оказаться метод `__cmp__()`.

Посмотрите, можно ли разработать ещё более простой способ выполнения метода `__setattr__()` в классе `egg` (см. листинг 13.6), заменив чем-либо конструкцию `try...except...else`.

Вы можете загрузить библиотеку `mayalib` с классами и функциями для манипуляций с датами и числами календаря Майя с Web-страницы, находящейся по адресу <http://www.pauahtun.org/TYPython/>. Интересные данные о календаре Майя Вы также найдёте на домашней странице по адресу <http://www.pauahtun.org/>.

14-й час

В предыдущей главе речь шла о специальных методах класса — одном из тех средств, которые делают Python особо дружественной средой объектно-ориентированного программирования (ООП). На этот раз мы собираемся применить ещё одну концепцию ООП — возможность повторного использования программных блоков. И хотя в текущем проекте мы не будем использовать классы, тем не менее, Вы в полной мере освоите все приёмы и методы разработки полноценных программных приложений. Не всё получится сразу, как нам хотелось бы, поэтому к нашей лаборатории "Франкенштейна" мы ещё вернёмся в следующей главе, чтобы воплотить в жизнь новые знания. Познакомившись с материалом этих двух глав, Вы научитесь создавать модули, специально предназначенные для повторного использования; документировать эти модули; добавлять в модули встроенные тесты; разбираться в кодах модулей, созданных другими программистами для общего пользования.

Создание программных компонентов для многократного использования

Всякий раз, когда Вам приходится создавать функцию или метод для выполнения той или иной рутинной задачи, которую наверняка Вам ещё не раз придётся решать в этом и последующих проектах, задумайтесь над созданием модуля многократного использования. Если такой модуль будет построен в соответствии с общепринятыми соглашениями, то его эффективно сможете использовать в своих программах не только Вы, но и другие программисты.

Пару лет тому назад я обнаружил, что некоторые из программ на языке Python, помещённые на моём Web-сервере, не работают. Я тщательно исследовал возникшую проблему и выяснил, что виной этому были неправильно записанные первые строки в файлах. Дело в том, что я работаю с сервером Apache Web в Windows NT. Считается, что такое решение позволяет настраивать Web-сервер и управлять им также, как в системах UNIX и Linux. В результате, хотя для всего мира мой сервер виден в среде NT, мне удалось зеркально отобразить его на две различные системы Linux (одна — на работе, другая — дома). Раздражающий недостаток состоит в том, что в версиях Apache для разных систем довольно много расхождений. Например, при копировании файлов страниц из системы в систему приходится вносить изменения в первых строках файлов для каждого интерфейса общего шлюза (CGI). Естественно, я часто забывал это делать, что приводило к серьёзным ошибкам. Пользователи не смогли пользоваться моим календарем Майя (<http://www.pauahtun.org/tools.html>), как я им это обещал, и слали мне гневные письма. Я решил

воспользоваться языком Python для написания коротенькой программы, которая автоматически вносила бы нужные изменения в соответствующие файлы.

***Прим. В. Шипкова: лично я предпочёл бы вынести все системные пути в виде строк в отдельный файл, и там, с помощью встроенных средств проверять ОС. В зависимости от результатов менял бы переменную. Для программы все эти махинации остались бы неизвестны. ;)**

Новый термин

Интерфейс CGI (Common Gateway Interface – интерфейс общего шлюза) – это способ, посредством которого приложения Web-серверов собирают информацию от пользователей. Для этой цели на Web-страницу обычно помещается специальная форма, а для обработки введённой в неё информации используется соответствующий сценарий CGI. Эти сценарии ещё иногда называют программами корзины CGI (cgi-bin program). Как правило, эти сценарии пишутся на языке Perl, иногда на C, но в последнее время все чаще для этой цели используется Python. В главе 24 мы рассмотрим несколько примеров сценариев CGI на языке Python.

Поскольку с такими проблемами могут столкнуться читатели этой книги, я решил предоставить код моей программы на обозрение публике. Начал я с достаточно простой программы, предназначавшейся для достижения всего лишь одной цели. Мы наделим её некоторой функциональностью, хотя, безусловно, для Вас останется ещё множество непройденных дорог для её совершенствования и расширения возможностей. Но те этапы, которые я намерен пройти за время планирования и создания данной программы, окажутся полезными и поучительными для Вас. Перед тем как приступить к работе, я сформулировал для себя следующий список вопросов.

- Что собой представляет проблема, которую я пытаюсь решить?
- Написал ли уже кто-нибудь программу, способную выполнить то, что мне надо?
- Каковы минимальные требования к тому, чтобы внесенные изменения можно было считать эффективными?
- Что нужно сделать, чтобы предупредить изменения данных в редактируемых файлах?
- Существуют ли модули, которые можно использовать в моей программе?

Проблема возникла из-за того, что сервер Apache в системе Windows или UNIX требует, чтобы файлы сценариев CGI,

которые не являются исполняемыми модулями (т.е. написаны не в машинном коде), должны начинаться строкой с двумя специальными символами (!), за которыми следует абсолютный путь к программе, ответственной за выполнение этого сценария. Для установленной у меня системы Red Hat Linux это означает, что первая строка любой программы на Python должна быть `#!/usr/bin/python`, а та же строка в системе NT принимает вид `#!c:\python\python.exe`. Другие системы UNIX и Linux часто помещают Python в папку `/usr/local/bin/`, благодаря чему в этих системах первая строка будет выглядеть как `#!/usr/local/bin/python`. Беглый поиск по Internet показал, что никто ещё не писал программ для решения подобных проблем, однако мне удалось обнаружить ссылку на модуль, который содержал как раз ту глобальную переменную, которая была мне нужна: `sys.executable`. Эта переменная содержит строку пути к интерпретатору Python. Таким образом, первые несколько строк моей программы приняли следующий вид:

```
import sys
pbang = "!" + sys.executable
if sys.platform == "Win32":
    sys.stderr.write ("Python Version "+
    sys.version+"Running on Windows\n")
elif sys.platform == "linux2":
    sys.stderr.write ( "Python Version " + \
    sys.version + " Running on Linux\n" )
else:
    sys.stderr.write ( "Python Version " + \
    sys.version + " Running on " + sys.platform + "\n" )
print pbang
```

Комбинацию символов `#!` в английской терминологии часто называют *hash-bang*, что на русский можно перевести приблизительно как "удар топором". Существует ещё множество полуофициальных сленговых названий для обозначения разных символов и их комбинаций. Не всегда можно понять, что именно навело то или иное название. Так, символ `*` иногда называют *shplat* (знать бы, что это такое), для символа `<` придумали *suck* (просто чмок), а для его побратима `>` – *spit* (плевок), пару фигурных кавычек `{` и `}` называют *squiggles* (локоны) и т.п. Требуемая информация о размещении программы Python на Вашем компьютере должна содержаться в переменной `pbang`. Чтобы вывести её на экран, проще всего сохранить приведённый выше код в файле `fixhash1.py` и запустить на выполнение, как показано на рис. 14.1.

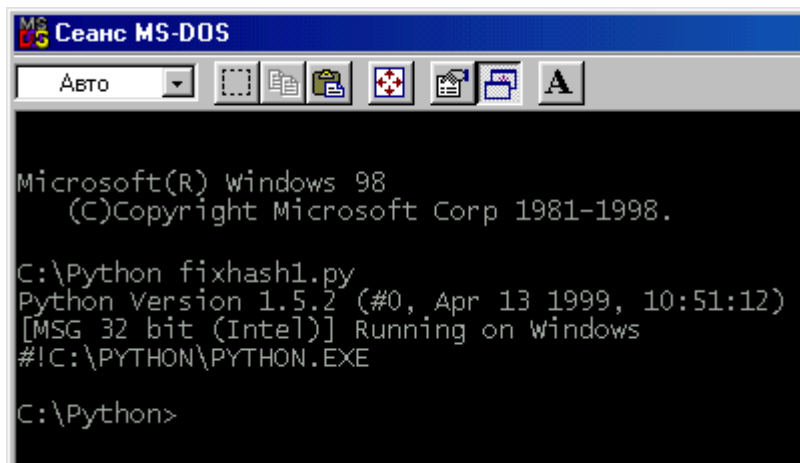


Рис. 14.1. Где живет Python?

В своей программе для вывода значения переменной `rbang` вместо обычной функции `print` я использовал функцию `sys.stderr.write()`, чтобы быть уверенным в том, что пользователи всегда смогут прочесть выводимое сообщение. Инструкция `print`, использующая по умолчанию объект `sys.stdout`, легко может быть перенаправлена в любое другое место. Если пользователь воспользуется командой `python fixhash.py > tempfile`, то после выполнения программы на экране не будет каких-либо сообщений о местонахождении Python и о том, что произошло с программой. Такой интерфейс программы нельзя назвать дружелюбным.

Объекты `stdin`, `stdout` и `stderr` унаследованы от соответствующих потоков ввода-вывода системы UNIX. В Python они автоматически открываются при запуске любой программы и являются объектами файлов. Эти имена означают стандартный ввод, стандартный вывод и стандартное сообщение об ошибке. В UNIX все эти объекты доступны программам в любой момент, чего нельзя сказать о Windows. Программы с родословной, уходящей корнями в UNIX, а к таковым относится и Python, должны использовать специальные функции, чтобы гарантировать доступность объектов ввода-вывода.

Поскольку я планировал использовать программу для изменения большого числа файлов, логично было в ядре программы заложить цикл, повторяющийся столько раз, сколько файлов нужно было обработать:

```
for i in список_файлов :  
    if i is not эта_программа :  
        проверить первую строку файла  
        if строка is not rbang:  
            изменение файла
```

Обратите внимание, что приведённый вверху листинг вовсе не программа на языке Python, а то, что мы обычно называем псевдокодом. Псевдокод представляет собой стенографический способ описания расширенной структуры программы. При планировании структуры программы вообще не обязательно тратить время на соблюдение всех требований синтаксиса программирования, поскольку псевдокод необходим для организации ваших мыслей, а не для выполнения на компьютере. Поэтому нет необходимости беспокоиться о возможных ошибках, лишь бы Вам стало ясно, как должна работать программа. Только после того как в псевдокоде программы станет ясно, что должен делать каждый блок и как он работает, приступайте к написанию настоящей программы. Написание программы обычно является циклическим процессом, поскольку в процессе создания и тестирования блоков программы часто возникают новые требования к программе и реализацию следует вновь начинать с написания псевдокода.

Первый шаг состоит в том, чтобы получить список файлов. Это довольно просто для тех, кто знает, что модуль `os` содержит функцию `os.listdir()`, специально предназначенную для этой цели. Следовательно, первые строки главной части программы должны быть такими:

```
dlist = os.listdir(".")
nfiles = 0
for i in dlist:
    if i != имя_программа
```

Как определить имя программы? Я категорически не хочу жёстко вносить в код имя файла, хотя это и было моим первым побуждением. Пришлось сделать усилие над собой, чтобы отказаться от первого подвернувшегося простейшего решения. Но тут я вспомнил, что модуль `sys` обеспечивает доступ к списку `argv`, содержащему все параметры, передаваемые при запуске программы. Первым элементом этого списка всегда является имя программы. Поскольку имя программы всегда содержит полный путь к выполняемому файлу, следует отбросить всю часть строки, предшествующую собственно имени файла. Это можно легко сделать следующим образом:

```
nm = os.path.split(sys.argv[0])[1]
```

Функция `os.path.split()` возвращает список, содержащий два элемента: путь и имя файла. Поэтому, чтобы присвоить переменной `nm` только имя файла, воспользуемся оператором индексирования и возвратим второй элемент списка (`[1]`). Строка пути останется в первом элементе списка (с индексом `0`). На этом этапе наш программный код имеет следующий вид:


```

nm = os.path.split (sys.argv[0])[1]
dlist = os.listdir ('.')
nfiles = 0
for i in dlist:
    if i == nm:
        continue
    ...проверка первой строки файла...

```

Один из способов проверки первой строки файла состоит в том, чтобы открыть этот файл, применив для этой цели функцию `open()`, и считать первую строку, после чего сравнить эту строку с той, которая представлена в переменной `pbang`. Однако следует учесть, что в будущем нам понадобится не только возможность проверки пути, но и возможность изменения его в файле. При работе с другими языками программирования, такими как С, чтобы внести изменения в первую строку файла, нужно перегибать не только её, но и весь файл. Такой подход требует создания временной копии файла, что само по себе неприятно. Представьте, что для изменения единственной строки Вам потребуется создание копии огромного файла. Это ли не пример крайней неэффективности работы программы. Я порылся в документации модуля `fileinput`, представленной по адресу www.python.org, и нашел то, что искал. Этот модуль содержит метод, позволяющий построчно считывать файл и тут же вносить изменения в текущую строку без создания временного файла. Вот он — ключ к достижению максимальной эффективности! Осталось написать функцию, которая будет переписывает первую строку файла, если в этом возникнет необходимость. Ядро программного кода приняло следующий вид:

```

nm = os.path.split (sys.argv[0])[1]
dlist = os.listdir (".")
nfiles = 0
for i in dlist:
    if i == nm:
        continue
    if len(i) > 4 :
        if i[-3:] == ".py":
            sys.stderr.write("Checking " + i + "...\\n")
            for line in fileinput.input(i):
                if fileinput.isfirstline():
                    if line[:-1] != pbang:
                        fileinput.close()
                        t = os.access(i,os.W_OK)
                        if t == 0:
                            sys.stderr.write(i + " is not writable;\\n skipping.\\n")
                        else:

```

```

        sys.stderr.write("Modifying " + i + \
            "...\\n" )
        rewritefile(i)
        nfiles = nfiles + 1
        break
    else:
        fileinput.close()
        break
sys.stderr.write("Files modified: " + 'nfiles' + "\\n")

```

Комментарий: модуль `os` содержит метод `access()`, который позволяет проверить, возможно ли открыть указанный файл для записи. Если файл разрешено открывать только для чтения, программа пропустит его, показав сообщение для пользователя. Имя `os.W_OK` представляет собой специальный флаг, сообщающий методу `access()`, что нам необходимо возвратить установку допуска для записи к этому файлу. Метод `isfirstline()` проверяет, работаем ли мы с первой строкой файла. Возвращенная первая строка сравнивается со значением переменной `rbang`. Если первая строка и значение `rbang` совпадают, следует команда: закрыть этот файл и перейти к следующему. Для этого используется инструкция `break` в конце программы, которая прерывает вложенный цикл `for`. Если же значения не совпадают, вызывается функция `rewritefile()` с именем текущего файла в качестве аргумента. Обычно перед сравнением имён файлов их приводят к нижнему регистру, но поскольку системы UNIX чувствительны к регистру, я решил, что лучше будет лишний раз переписать первые строки файлов в Windows, чем пропустить хотя бы одну несовпадающую запись в UNIX. Последняя строка сообщает пользователю, сколько файлов было изменено в ходе выполнения программы.

Чтобы проверить правильность кода, можно временно заменить функцию `rewritefile()` на так называемую функцию-заглушку (*stub-function*). Это функция, имеющая ту же сигнатуру (набор параметров), что и заменяемая функция, но в то же время не выполняющая никаких действий или просто выводящая сообщение для отладки программы. Например, можно создать функцию, которая будет выводить имена файлов текущего каталога. Вставьте эту функцию в файл `fixhash.py` и запустите программу на выполнение, чтобы проверить правильность её работы.

Ниже показан настоящий код функции `rewritefile()`.

```

def rewritefile(fip):
    mode = os.stat(fip)[ST_MODE]

```



```

for line in fileinput.input(fip, inplace=1):
    if fileinput.isfirstline():
        if line[:2] == "!!" :
            sys.stdout.write(pbang + "\n")
        else:
            sys.stdout.write(pbang+"\n")
            sys.stdout.write(line)
    else:
        sys.stdout.write(line)
fileinput.close()
os.chmod(fip, mode)

```

Комментарий: поскольку мы имеем дело с функцией, в ней есть ссылки на переменные, определённые в модуле, но функция не может присваивать, им какие-либо значения. Так, мы не можем изменить значение переменной `rbang`, если только явно не откроем к ней доступ с помощью инструкции `global rbang`. Но значение этой переменной должно устанавливаться только однажды по контексту открытия модуля, после чего оно остаётся константным. Поэтому невозможность изменения переменной `rbang` функциями модуля вполне нас удовлетворяет. Модуль `os` содержит функцию `stat()`, которая возвращает всю информации о состоянии файлов и каталогов. Нас в данном случае интересует состояние открываемых файлов. Но чтобы получить определение `ST_MODE`, следует импортировать модуль `stat`. В данном случае вполне подойдёт выражение `from stat import *`. В действительности нас совершенно не интересуют текущие состояния файлов, просто после их изменения следует восстановить исходные установки. По флагам состояния файла можно определить, является ли он исполняемым и можно ли его открывать для записи или только для чтения. В ходе изменения содержимого файла часто автоматически изменяются установки флагов состояния, поэтому перед закрытием файла необходимо восстановить его исходное состояние. В действительности это важно только для системы UNIX и совершенно не важно для Windows. Но в целях поддержания переносимости программы всегда следует запоминать и восстанавливать флаги состояния изменяемых файлов.

Выражение `for line in fileinput.input(fip, inplace=1):` основано на одном исключительно полезном свойстве модуля `fileinput`. Ключевой аргумент `inplace=1` задаёт особый режим открытия файла, позволяющий тут же вводить изменения в текущие строки. При вызове функции `fileinput.input()` с данным аргументом автоматически создаётся копия файла, после чего он открывается повторно в режиме для записи. С этого момента любые вызовы функции `write()` или инструкции `print`, которые связаны с объектом `stdout`, передают данные в новый файл. После открытия файла программа проверяет его

первую строку. Если строка начинается с символов #!, то она замещается на стандартную (содержащуюся в переменной pbang). Если же файл начинается как-то иначе, то стандартная строка добавляется в начало файла, после чего все содержимое файла, строка за строкой, передаётся в объект stdout, который тут же перенаправляет их в новый файл. Когда происходит вызов функции fileoutput.close(), новый файл закрывается, а копия исходного файла удаляется из оперативной памяти компьютера.

В результате мы получили новую версию программы fixhash.py, код которой показан в листинге 14.1.

Листинг 14.1. Завершенная программа fixhash.py

```
#!c:\Spez\Python-2-4\python.exe

import os
import sys
import fileinput
from stat import *

pbang = "#!" + sys.executable
if sys.platform == "win32" :
    sys.stderr.write ( "Python Version "+sys.version+"\n"
        Running on Windows\n" )
elif sys.platform == "linux2":
    sys.stderr.write("Python Version "+sys.version+"\n"
        Running on Linux\n" )
else:
    sys.stderr.write ( "Python Version "+sys.version+"\n"
        Running on " + sys.platform + "\n" )

nm = os.path.split ( sys.argv[ 0 ] )[ 1 ]

def rewritefile ( fip ) :
    global pbang
    mode = os.stat ( fip )[ ST_MODE ]
    for line in fileinput.input ( fip, inplace = 1 ) :
        if fileinput.isfirstline():
            if line [ : 2 ] == "#!" :
                sys.stdout.write ( pbang + "\n" )
            else:
                sys.stdout.write ( pbang + "\n" )
                sys.stdout.write ( line )
        else:
            sys.stdout.write ( line )
    fileinput.close ( )
    os.chmod ( fip, mode )
```

```

dlist = os.listdir ( "." )
nfiles = 0
for i in dlist :
    if i == nm :
        continue
    if len ( i ) > 4 :
        if i[ -3 : ] == ".py" :
            sys.stderr.write ( "Checking " + i + "...\\n" )
            for line in fileinput.input ( i ) :
                if fileinput.isfirstline ( ) :
                    if line[ : -1 ] != pbang :
                        fileinput.close ( )
                        t = os.access ( i, os.W_OK )
                        if t == 0 :
                            sys.stderr.write ( i + " is not writable;\\n
                                skipping.\\n" )
                        else :
                            sys.stderr.write("Modifying "+i+"...\\n" )
                            rewritefile ( i )
                            nfiles = nfiles + 1
                            break
                    else :
                        fileinput.close ( )
                        break
sys.stderr.write("Files modified: "+`nfiles`+"\\n" )

```

Представленная выше программа fixhash.py – это законченное работоспособное приложение, превосходно справляющееся с поставленной задачей. Никакие другие изменения не потребуются до тех пор, пока с ней будете работать только Вы сами. Если же Вы хотите превратить её в модуль, который сможете использовать Вы и другие программисты в качестве компонента своих программ, следует ещё немножко поработать.

Самой первой полезной модернизацией будет добавление аргумента командной строки, задающего текущий каталог. Это позволит Вам сохранить файл fixhash.py в общем каталоге, где хранятся все файлы сценариев, вместо того чтобы копировать этот файл каждый раз в тот каталог, который нужно проанализировать. В системе UNIX таким общим местом хранения файлов сценариев обычно является каталог bin, вложенный в другой общедоступный каталог, например /usr/local/bin. В Windows, однако, стандартных папок для этих целей не существует. Я на своём компьютере использую инструментальный набор средств MKS для Windows, позволяющий воспроизвести среду UNIX. В этом случае для сценариев

общего доступа рекомендуется использовать каталог mksnt (для любой версии Windows это обычно c:\mksnt). Вы можете выбрать любую папку на Вашем компьютере, наиболее удобную для Вас.

Если проанализировать главную часть программы, то совсем не трудно определить, какие именно изменения необходимо внести, чтобы стала возможной поддержка аргумента командной строки. В листинге 14.2 показаны необходимые изменения, которые свелись к определению новой функции fixdirectory и добавлению блока

```
if __name__ == "__main__" :  
    ....
```

Листинг 14.2. Добавление поддержки аргумента командной строки в программу fixhash.py

```
def fixdirectory(d) :  
    dlist = os.listdir ( d )  
    nfiles = 0  
    for i in dlist :  
        if i == nm :  
            continue  
        if len ( i ) > 4 :  
            if i[ -3 : ] == ".py" :  
                sys.stderr.write ( "Checking " + i + "__\n" )  
                for line in fileinput.input ( i ) :  
                    if fileinput.isfirstline ( ) :  
                        if line[ :-1] != pbang :  
                            fileinput.close ( )  
                            t = os.access ( i, os.W_OK )  
                            if t == 0 :  
                                sys.stderr.write(i+\n  
                                " is not writable;skipping. \n")  
                            else :  
                                sys.stderr.write ( "Modifying " + i + "... \n"  
                                rewritefile ( i )  
                                nfiles = nfiles + 1  
                                break  
                        else :  
                            fileinput.close ( )  
                            break  
    return nfiles  
  
if __name__ == "__main__" :  
    nmod = 0  
    if len ( sys.argv ) > 1 :
```

```

    dirnames = sys.argv[ 1 : ]
else :
    dirnames = [ "." ]
    for s in dirnames :
        nmod = nmod + fixdirectory ( s )
sys.stderr.write ( "Files modified: " + 'nmod' + "\n" )

```

После внесения указанных изменений я запустил программу `fixhash.py` на выполнение сначала из текущего каталога, при этом никаких проблем не возникало, а после этого из общего каталога сценариев с указанием проверяемого каталога `C:\DB` в аргументе командной строки. На рис. 14.2 показано, что произошло во втором случае.

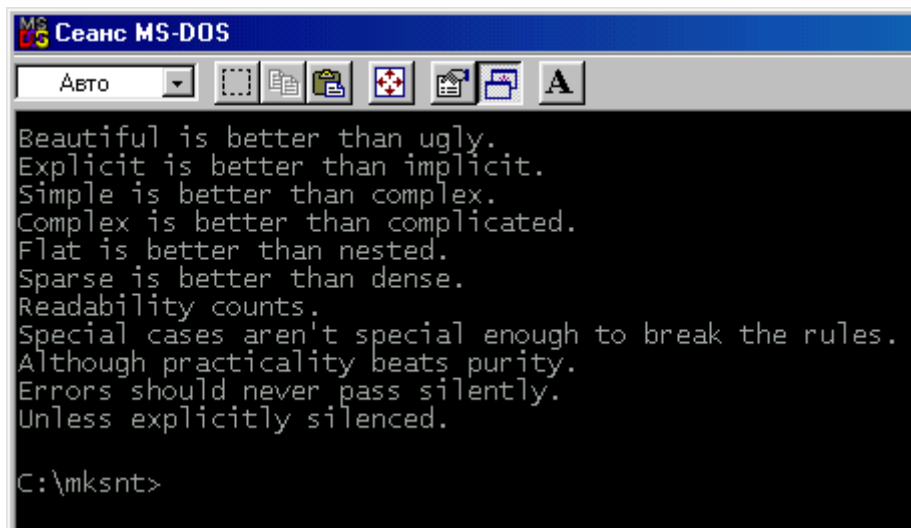


Рис. 14.2. Приехали?!!

А-а-а, я забыл одну важную вещь (честное слово, со мной это случается не часто). Аргумент командной строки работает нормально, однако, несмотря на то, что программа находит нужный каталог (в противном случае откуда она знает, какой файл предполагается открыть), нужный файл она не может открыть. Это объясняется тем, что модулю `fileinput` передаётся только имя файла, но не путь к нему. Это нелегко исправить, как показано в листинге 14.3.

Листинг 14.3. Внесение исправлений в файл `fixhash.py`

```

def fixdirectory ( d ) :
    dlist = os.listdir ( d )
    nfiles = 0
    for i in dlist:
        if i == nm:
            continue
        if len ( i ) > 4 :
            fname = d + "7" + i

```

```

if fname[ -3 : ] == ".py" :
    sys.stderr.write ( "Checking " + fname + "...\\n" )
    for line in fileinput.input ( fname ) :
        if fileinput.isfirstline ( ) :
            if line[ :-1] != pbang:
                fileinput.close ( )
                t = os.access { fname, os.W_OK }
                if t == 0:
                    sys.stderr.write ( fname + " is not\\
                        writable; skipping.\\n" )
                else:
                    sys.stderr.write ( "Modifying " + fname +\\
                        ...\\n" )
                    rewritefile ( fname )
                    nfiles = nfiles + 1
                    break
            else:
                fileinput.close ( )
                break
    return nfiles

```

Если теперь мы запустим программу снова, то получим правильный результат, как показано на рис. 14.3.

Рис. 14.3. Теперь с помощью программы *fixhash.py* можно изменять файлы в любом каталоге

Обратите внимание, что даже в Windows нет необходимости использовать в строке пути обратные косые. Часто приходится ломать голову, какую косую черту использовать. Но Python

автоматически преобразует обычную косую черту в обратную, если ваша операционная система потребует это.

А что дальше? Мы успешно добавили аргумент командной строки. Между прочим, цикл `for s in dirnames:` (строка 32 в листинге 14.2) обеспечивает возможность того, что несколько имён каталогов, указанных в командной строке, будут обрабатываться правильно. Но можно ли ещё как-то расширить функциональные возможности программы `fixhash.py`? Было бы здорово, если бы функция `fixdirectory()` искала файлы Python не только в указанном каталоге, но и во всех вложенных каталогах. Такой поиск файлов называется рекурсивным.

Это совсем не трудно реализовать. Я внимательно просмотрел информацию обо всех доступных модулях в документации на Python, и моё внимание привлекла функция `path.walk()` из модуля `os`. Она оказалась именно тем, что нам нужно. Поскольку мы уже пользуемся функцией `path.split()` из модуля `os`, нам даже не нужно импортировать никакие новые модули. Кроме того, в этом модуле есть ещё одна полезная функция `path.join()`, с помощью которой удобно выполнять конкатенацию строк там, где это нужно было в листинге 14.3. Функция `path.join()` настолько сообразительна, что может даже самостоятельно в строке пути расставлять недостающие косые. Так, запись `fname=d+"/"+i` будет равносильна выражению: `fname= os.path.join(d,i)`

Если имя каталога `d` в рассматриваемом примере оканчивается косой чертой, функция не станет добавлять ещё одну косую, в то время как обычный оператор конкатенации строк именно так и поступит.

В данный момент функция `fixdirectory()` принимает только один аргумент, но этого будет недостаточно, если мы хотим добавить функцию `os.path.walk()`. Ниже показан синтаксис этой функции: `walk(path,visit, arg)`

А вот что говорится об этой функции в документации Python: "Вызывает функции `visit` с аргументами (`arg`, `dirname`, `names`) для каждого каталога в дереве каталогов, заданном аргументом `path` (путь), включая базовый каталог, заданный путём. Параметр `dirname` (имя каталога) определяет текущий подкаталог; параметр `names` (имена) представляет собой список файлов в этом каталоге, полученный в результате выполнения функции `os.listdir(dirname)`".

Чтобы воспользоваться функцией `walk()`, нужно только вызвать её в главной части программы, передав имя текущего каталога (аргумент `path`), скорректированную нами версию функции `fixdirectory()` (аргумент `visit`) и ещё один аргумент

arg. Последний аргумент просто задаёт способ передачи данных функции visit. Обычно в этом аргументе нет необходимости, но при некоторых обстоятельствах он может оказаться весьма удобным средством модернизации функции от дерева к дереву. В листинге 14.4 представлен изменённый код функции fixdirectory().

Листинг 14.4. Изменённая версия функции fixdirectory()

```
def fixdirectory ( arg, d, dlist ):
    global nfiles
    nfiles = 0
    for i in dlist:
        if i == nm:
            continue
        if len ( i ) > 4:
            fname = os.path.join ( d, i )
            if fname[-3:] == ".py":
                sys.stderr.write ( "Checking "\
                + fname + "...\\n" )
                for line in fileinput.input(fname):
                    if fileinput.isfirstline():
                        if line[:-1] != pbang:
                            fileinput.close()
                            t = os.access(fname, os.W_OK)
                            if t==0:
                                sys.stderr.write(fname + " is not\
                                writable;skipping.\\n" )
                            else:
                                sys.stderr.write("Modifying "+fname +\
                                "...\\n")
                                rewritefile(fname)
                                nfiles=nfiles+1
                                break
                    else:
                        fileinput.close()
                        break
```

Обратите внимание, как мало требуется внести изменений. Просто добавьте новые параметры, удалите обращение к функции os.listdir() и добавьте строку global nfiles. Последнее необходимо для того, чтобы определить число изменённых файлов. Для определения этого числа в наш модуль нужно включить дополнительную функцию:

```
def getnfiles():
    return nfiles
```


Разумеется, можно было бы рекомендовать пользователям определять число файлов посредством прямого доступа к переменной `fixhash.nfiles`, но, пожалуй, интерфейс будет менее сложным, если мы предусмотрим для этой цели простую функцию.

Ещё нужно внести небольшие изменения в основную часть программы, как показано в листинге 14.5.

Листинг 14.5. Изменение программы `fixhash.py` с целью поддержания использования функции `os.path.walk()`

```
if __name__ == "__main__":
    nmod = 0
    if len( sys.argv )> 1:
        dirnames = sys.argv[1:]
    else:
        dirnames=["."]
    for s in dirnames:
        os.path.walk(s,fixdirectory,0)
        nmod=nmod+getnfiles()
    sys.stderr.write("Files modified:\n"
        + 'nmod' + "\n" )
```

На рис. 14.4 показан результат выполнения нашей модифицированной программы. В качестве аргумента был передан каталог, содержащий вложенный подкаталог, причём в обоих хранились файлы Python.

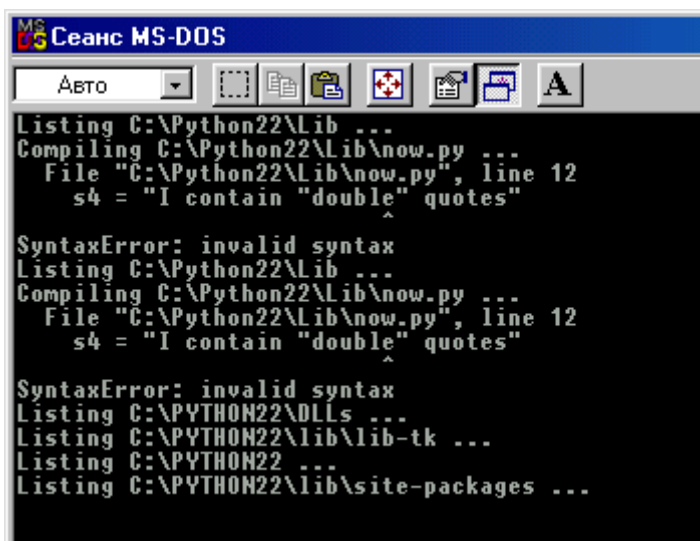


Рис. 14.4. Просмотр дерева каталогов программой `fixhash.py`

Резюме

В этой главе мы рассмотрели все этапы разработки модулей для многократного использования. Был создан модуль `fixhash`, который мы затем многократно совершенствовали, добавляя всё новые и новые возможности. В следующей главе мы завершим создание этого модуля и выполним его документирование.

Практикум

Вопросы и ответы

Почему Python устанавливается в разных каталогах в различных версиях UNIX?

Основная причина заключается в различии двух главных ветвей системы UNIX: BSD и System V. Изначально не было предусмотрено специальных мест для хранения пользовательских сценариев и программ. Но их с каждым годом накапливалось всё больше и больше. В силу этого обстоятельства компания Berkeley (BSD) выделила специальный каталог `/usr/local`. Позднее подобные каталоги были зарезервированы и в System V, причём были разработаны чёткие правила, регламентирующие размещение программ в этих каталогах. Различные версии системы Linux добровольно приняли на себя обязательство использовать некоторые или даже все эти каталоги как системы BSD, так и системы System V.

Контрольные вопросы

1. Какой из предлагаемых ниже методов возвращения списка файлов в каталоге является наилучшим?
 - а) `os.path.list()`
 - б) `os.listdir()`
 - в) `sys.listdir()`
 - г) `sys.dir()`
2. Для чего используется переменная `sys.platform`?
 - а) Сообщает об операционной системе, с которой Вы работаете.
 - б) Сообщает о платформе Вашего компьютера.
 - в) Сообщает о том, что Вы работаете с Active Desktop.
 - г) Такой переменной нет.

Ответы

1. б. Наилучший и, пожалуй, единственный из представленных способов (если Вы, конечно, не напишете свой метод) —

воспользоваться методом `os.listdir()`. Функция `dir()` решает совсем другую задачу.

2. а. Переменная `sys.platform` сообщает Вам о том, на какой операционной системе выполняется ваша программа. Это позволит Вам избежать использования функций, которые недоступны в данной операционной системе.

Примеры и задания

Великолепный пример формы, принимающей данные от пользователей, можно посмотреть по адресу <http://www.ithaca.edu/library/Training/wwwquiz.html>. На этой же странице Вы найдёте вполне приличный словарь терминов Internet.

Попробуйте самостоятельно создать функцию-заглушку для подмены в нашей программе функции `rewritefile()`. Постарайтесь сделать так, чтобы программа не только работала, но и добавленная Вами функция могла сообщать полезную информацию об ошибках во время отладки программы.

В строке 32 листинга 14.1 устанавливается поиск в текущем каталоге всех файлов, оканчивающихся стандартным для Python расширением `.py`. Подумайте, что произойдет, если кто-нибудь присвоит вложенному каталогу имя типа `spam.py`. Найдите в документации по Python описание способа определить, является ли некоторый файл действительно файлом, а не каталогом. Добавьте соответствующий программный код в нашу программу `fixhash.py`. Подсказка: прочтите документацию к функции `os.stat()`.

Разработайте вариант функции `fixdirectory()`, нечувствительной к регистру букв. Попробуйте сделать так, чтобы ваша функция "теряла чувствительность" к регистру в системе Windows, но сохраняла её в других системах. Подсказка: просмотрите документацию к модулю `string`.

15-й час

Лаборатория доктора Франкенштейна (продолжение)

В предыдущей главе мы приступили к созданию модуля многократного использования. Мы уже хорошо потрудились над его функциональностью, но чтобы получить действительно профессионально сработанный модуль, осталось выполнить ещё ряд действий. Этим мы и займёмся в данной главе. Мы закрепим навыки проектирования программных продуктов, кроме того Вы узнаете, как построить модуль, специально предназначенный для повторного использования; выполнить

документирование модуля; включить в модуль средства тестирования; использовать в модуле блоки, разработанные другими программистами.

Создание программных компонентов многократного использования (продолжение)

В конце предыдущей главы мы получили работающую версию модуля `fixhash.py`, который можно запустить в работу из любого каталога с указанием в командной строке того каталога, который нужно протестировать. Более того, программа автоматически проанализирует все каталоги, вложенные в текущий.

Впрочем, будущий пользователь может как раз захотеть, чтобы программа не выполняла рекурсивный поиск файлов. (Рекурсивным называется такой поиск, который начинается с заданного каталога с последующим посещением всех вложенных подкаталогов, принадлежащих текущему дереву каталогов. Существуют и другие понятия рекурсии, но в данном случае мы имеем в виду только способ поиска файлов). Чтобы предоставить пользователю возможность выбирать схему поиска, добавим ещё один параметр командной строки, который будет включать или выключать рекурсивный поиск. В листинге 15.1 показан новый вариант ядра программы.

```
if __name__ == "__main__" :
    nmod=0
    recurse=0
    if len(sys.argv)>1:
        if sys.argv[1]=='-r':
            recurse=1
            if recurse and len(sys.argv)>2:
                dirnames=sys.argv[2:]
            elif recurse and len(sys.argv)<2:
                dirnames=["."]
            else:
                dirnames=sys.argv[1:]
            else:
                dirnames=["."]
    for s in dirnames:
        if not recurse:
            dlist=os.listdir(s)
            fixdirectory(0,s,dlist)
        else:
            os.path.walk(s,fixdirectory,0)
            nmod=nmod+getnfiles()

    sys.stderr.write ( "Files modified: " + 'nmod' + "\n" )
```

На рис. 15.1 показан результат выполнения вызова - модуля без аргументов (по умолчанию): `python fixhash.py`, а на рис. 15.2 представлен результат выполнения того же модуля с установкой аргумента рекурсии и точкой после него для указания анализа текущего каталога: `python fixhash.py -r ...`

В листинге 15.2 показан окончательный вариант модуля `fixhash.py`.



Рис. 15.1. Вызов модуля по умолчанию: `fixhash.py`



Рис. 15.2. Вызов модуля с аргументом рекурсии: `fixhash.py -r`

Листинг 15.2. Окончательный вариант модуля `fixhash.py`

```
#!/c:\python\python.exe

import os
import sys
import fileinput
from stat import

pbang="#!" + sys.executable
nfiles = 0
nm=""

def initial(v=0):
    global nm
    if v!=0:
        if sys.platform=="Win32":
            sys.stderr.write("Python Version "+\
                sys.version+" Running on Windows\n")
        elif sys.platform=="linux2":
            sys.stderr.write("Python Version "\
                +sys.version+" Running on Linux\n")
```

```

else:
    stderr.write("Python Version "+sys.version+" \
Running on "+sys.platform+"\n")
    nm=os.path.split(sys.argv[0])[1]

def getnfiles():
    return nfiles

def rewritefile(fip):
    global pbang
    mode = os.stat(fip) [ST_MODE]
    for line in fileinput.input(fip, inplace=1):
        if fileinput.isfirstline():
            if line[:2]=="#!":
                sys.stdout.write(pbang+"\n")
            else:
                sys.stdout.write(pbang+"\n")
                sys.stdout.write(line)
        else:
            sys.stdout.write(line)
            fileinput.close()
            os.chmod(fip,mode)

def fixdirectory(arg, d, dlist):
    global nfiles
    nfiles = 0
    for i in dlist:
        if i==nm:
            continue
        if len(i)>4:
            fname=os.path.join(d, i)
            if fname[-3:]=="py":
                if arg!=0:
                    sys.stderr.write("Checking"+fname+"\
...\n" )
                    for line in fileinput.input(fname):
                        if fileinput.isfirstline():
                            if line[:-1]!=pbang:
                                fileinput.close()
                                t=os.access(fname, os.W_OK)
                                if t==0:
                                    if arg!=0:
                                        sys.stderr.write(fname+" is not\
writable; skipping.\n")
                                else:
                                    if arg!=0:
                                        sys.stderr.write("Modifying " +\
fname + "...\n")

```

```

        rewritefile(fname)
        nfiles=nfiles+1
        break
    else:
        fileinput.close()
        break
    return nfiles

def fixhash(d, r, v):
    nmod=0
    if type(d)!=type([]):
        dl=list(d)
    else:
        dl=d
    initial(v)
    for i in dl:
        if not r:
            dlist=os.listdir(i)
            fixdirectory(v, i, dlist)
        else:
            os.path.walk(i, fixdirectory, v)
            nmod=nmod+getnfiles()

    return nmod

if __name__=="__main__":
    nmod=0
    recurse=0
    if len(sys.argv)>1:
        if sys.argv[1]=='-r':
            recurse=recurse+1
            if recurse and len(sys.argv)>2:
                dirnames=sys.argv[2:]
            elif recurse and len(sys.argv)<2:
                dirnames=["."]
        else:
            dirnames=sys.argv[1:]
    else:
        dirnames=["."]
    nmod=fixhash(dirnames, recurse, 1)

    sys.stderr.write ( "Files modified: " + 'nmod' + \
        "\n" )

```

Обратите внимание, что заключение выполняемой части программы в конструкцию `if __name__=="__main__":` даёт возможность пользователю одинаково успешно выполнять файл `fixhash.py` как уже готовую программу либо импортировать

этот модуль в свою программу. В листинге 15.3 показано, как выполняется импортирование.

Листинг 15.3. Программа testhash.py

```
#!c:\python\python.exe

import sys
import fixhash

if __name__=="__main__":
    nmod=0
    recurse=0
    verbose=0
    if len(sys.argv)>1:
        if sys.argv[1]=='-r':
            recurse=recurse+1
            if recurse and len(sys.argv)>2:
                dirnames=sys.argv[2:]
            elif recurse and len(sys.argv)<2:
                dirnames=["."]
            else:
                dirnames=sys.argv[1:]
        else:
            dirnames=["."]
            nmod=fixhash.fixhash(dirnames, recurse, \
                                verbose)

    if verbose:
        sys.stderr.write("Files modified: '"+nmod'\n"
                        +"\n")
    else:
        sys.exit(nmod)
```

Документирование пользовательских модулей

Одной из важных задач, которую Вам предстоит ещё выполнить для усовершенствования самого модуля и повышения возможности его эффективного использования пользователями, является документирование того, что и как он может делать. Это минимальные требования к документации модуля. Я предпочитаю включать ещё описания возможных ошибок, а также прилагать документацию в формате HTML. Действительно, нет проблем скопировать строки документации в Ваш редактор HTML и сохранить их отдельным файлом, доступным для просмотра в Internet.

Вы убедитесь в том, что модули, снабжённые подробной документацией, пользуются значительно большим спросом и

популярностью, чем те, с особенностями которых программистам приходится разбираться самостоятельно. Не так давно я собрал воедино разработанные мной сценарии оболочки, которые, на мой взгляд, казались весьма полезными и вполне понятными, поэтому я не счел нужным подготовить соответствующую документацию на них. Я объявил по своему списку рассылки о том, где любой желающий может получить доступ к моим утилитам. Уже на следующий день я получил три письма, сообщавших, что мои разработки, вполне вероятно, весьма интересны и полезны, но в них совершенно нельзя разобраться непосвященному. Я не стал ожидать следующих сообщений с критикой и быстро составил необходимую документацию. Сделал я это действительно быстро. До этого гораздо больше времени я потратил на размышления, стоит документировать мои модули или не стоит.

Вместо того чтобы вновь приводить здесь весь код нашей программы `fixhash.py` с добавленными строками документации, рассмотрим лучше в качестве примера документацию лишь одной функции `fixhash` (листинг 15.4). Подобные строки с описанием возможностей и особенностей применения можно добавить во все функции, а также в первые строки самого модуля. Займитесь этим самостоятельно в качестве домашнего задания.

Листинг 15.4. Документирование функции `fixhash`

```
def fixhash ( d, r, v ) :
    """
    Данная функция вызывает функцию fixdirectory ( ) для
    каждого каталога, указанного в списке d. Если r не равно
    нулю, функция os.path.walk() выполняется в рекурсивном
    режиме поиска файлов в дереве каталогов, начиная с
    корневого каталога. Если v не равно нулю, то выводятся
    сообщения об ошибках. Возвращает количество измененных
    файлов.
    """
    nmod=0
    if type(d)!=type([]):
        dl=list(d)
    else:
        dl=d
    initial(v)
    for i in dl:
        if not r:
            dlist=os.listdir(i)
            fixdirectory(v, i, dlist)
        else:
            os.path.walk(i, fixdirectory, v)
            nmod=nmod+getnfiles()
```

```
return nmod
```

На рис. 15.3 показан пример документирования функций в формате HTML. С полной документацией на английском языке всех модулей и функций, использованных в данной книге, можно по адресу <http://www.pauahtun.org/TYPython/>

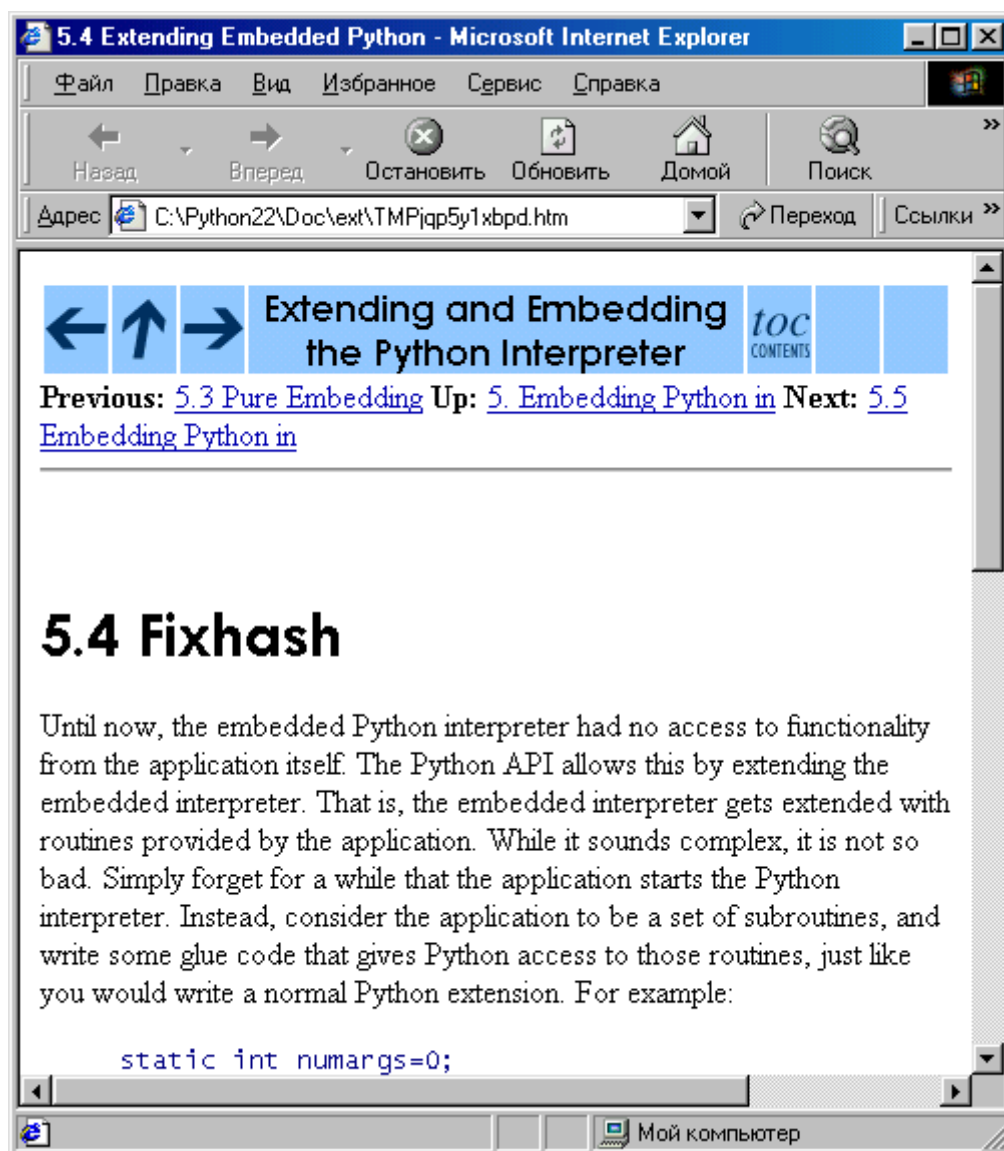


Рис. 15.3. Пример документации функций модуля *fixhash.py* в формате HTML

Следует отличать в Python строки документации от строк комментариев в программе. Оба средства весьма эффективны, но служат для разных целей, а потому к ним предъявляются разные требования. Помню, когда я только осваивал премудрости программирования, то часто обращался за помощью к более опытному инженеру, с которым работал в одном отделе. Сначала я сопровождал свой программный код очень подробными комментариями, считая, что тем самым делаю его максимально понятным. Мой товарищ долго рассматривал

программу, задумчиво теребя пальцами бороду, потом сказал, что в ней трудно разобраться, поскольку тут так много комментариев, что в них теряется код программы. Я стал спорить, что комментарии для того и придумали, чтобы на "человеческом" языке разъяснить, что делает программа в том или ином месте. Тогда он просто переписал мою программу, безжалостно сокращая комментарии до минимума. Ещё до того как он закончил трудиться над моей программой, я уже смог обнаружить свои ошибки в коде.

Если комментарии должны быть ёмкими, то в документации Вы можете позволить себе подробные описания, естественно, в разумных пределах. Документация размещается в первых строках программы, поэтому не мешает проследить логику выполнения программного блока. Кроме того, строки документации можно вывести на экран для просмотра отдельно от программного кода.

При написании комментариев не следует указывать, что делает каждая строка программного кода, тем более, когда это очевидно. С их помощью лучше выделить ключевые выражения кода, описать способ вызова функции и что она возвращает. Так, следующий комментарий совершенно бессмыслен, так как очевиден:

```
i=i+1 # Увеличить значение i на единицу
```

Если весь Ваш код будет пестреть подобными комментариями, то это приведёт в бешенство кого угодно.

Лучше всего будет дать описание функции в документации, а затем снабдить краткими комментариями ключевые моменты её выполнения, как показано на примере в листинге 15.4.

Листинг 15.4. Пример использования документации и комментариев

```
def FindYear(s):
    """Поиск строки по значению года, заданному в формате
    "(1999)" с пробелом впереди. Возвращает строку без значения
    года, за которой следует год в числовом формате. Если год
    не указан, возвращается None.
    """
    global _yearre
    global _yearrec
    if _yearrec==0:
        _yearrec=re.compile(_yearre)
        so=_yearre.search(s)
        if so==None:
```

```
    return None
    st, ep =so.span()
    st=st+" Удаление пробела"
    y=string.atoi(s[st+1:ep-1])
    rt=s[:st-1] + s[ep:]
    return rt, y
```

Комментарий всего лишь привлекает Ваше внимание к строке программного кода, смысл которой Вам может показаться недостаточно очевидным: "С помощью строки 15 мы просто удаляем нежелательный пробел".

В других языках строки комментариев часто используются для документирования программных блоков, функций, классов и др. Как уже говорилось, в Python для этих целей удобнее добавлять в программный код строки документации. Хотя, если Вы к этому привыкли, то можете и дальше использовать строки комментариев для документации блоков программы. Особенно это полезно в тех случаях, если Вы используете в своей программе чужие плохо задокументированные модули. В этом случае комментарии, как пометки на полях, помогут Вам лучше разобраться в незнакомом коде.

Тестирование программных компонентов

Наверное, Вы уже убедились, насколько важно бывает для правильного выполнения программы определить текущее значение или состояние переменной, флага или модуля. Наш модуль `fixhash` имеет программный блок для самотестирования, следующий за инструкцией:

```
if __name__ == "__main__":
```

(см. строку 89 в листинге 15.2). Как мы помните, данный блок инструкции `if` выполняется только в случае запуска модуля как самостоятельной программы, но не будет выполняться при импортировании этого модуля в другую программу. Без блока тестирования часто оказывается просто невозможно разобраться с тем, как работает программа и почему в импортированном модуле происходит сбой. Иногда строки тестирования вводят временно, только на время отладки, иногда их оставляют в коде программы навсегда, если это будет полезным с точки зрения будущих пользователей.

Ниже приведён список ситуаций, когда выполнение тестирования будет полезным для отладки или выполнения программы.

- Тестирование граничных условий. Например, если некоторая функция принимает числовые значения в заданном диапазоне, проверьте, как будет работать эта функция при передаче ей значений, входящих в диапазон, за пределами диапазона и граничных значений.
- Тестирование ошибочного ввода. Например, если функция ожидает строку цифр, проверьте, как она себя поведет, если ей передать строку букв.
- Тестирование контекста вызова модуля. С этим тестированием мы уже знакомы. Его следует выполнять во всех модулях, использование которых предполагает импортрование.
- Тестирование стабильности модуля. Попробуйте присвоить из вызывающей программы переменным импортированного модуля непредусмотренные значения. Например, что произойдет, если переменной `st` модуля `fixhash` присвоить символ точки "." или числовое значение.
- Тестирование зависимости от размещения. Выполните Вашу программу из другого каталога, отличного от того, в котором Вы его создали. Если ожидается ввод текущего каталога из командной строки, попробуйте передать ей абсолютный и относительный пути.
- Тестирование ошибкоустойчивости. Внимательно проверяйте и по возможности избавляйтесь от программных блоков, которые не внушают Вам доверия. Они могут оказаться минами замедленного действия и сработать именно в тот момент, когда Вы будете передавать программный продукт заказчику.
- Тестирование самого себя. Запомните раз и навсегда, что клиент всегда прав. Если у пользователя возникли проблемы с вашей программой из-за того, что он ввёл не те данные, то виноваты в этом Вы сами. Во время тестирования программы смотрите на неё не как разработчик, а как вредный пользователь.

Поиск модулей независимых изготовителей для своих программ

На примере модуля `fixhash` я пытался максимально подробно раскрыть процесс разработки программного блока многократного использования. Но прежде чем приступить к разработке своего модуля, нужно ответить на следующие два вопроса. Во-первых, разрабатывал ли кто-либо до Вас подобные модули? Во-вторых, существуют ли готовые модули, которые могут быть использованы в качестве компонентов Вашего модуля? Как обычно, наиболее подходящим местом, где можно найти информацию такого рода, является домашняя страница Python (<http://www.python.org/>). На этой Web-странице Вы можете воспользоваться средством поиска по ключевым словам. Таким способом можно провести поиск по

серверу Python, серверам телеконференций или по всей сети Internet. В телеконференциях представлено много сообщений о новых программных средствах, приёмах, идиомах и справочных пособиях. Самое важное для Вас — знать, что, где и как искать. Тема поиска информации и программных продуктов в Internet выходит за рамки этой книги. Много полезной и справочной информации Вы найдёте на странице, размещенной по адресу <http://www.python.org/search/>. Если Вы не сильны в средствах поиска, начните свою охоту на модули с этой страницы, которая популярно объяснит Вам, как правильно расставить свои сети в Internet.

Если Вы зарегистрируетесь в списках рассылки пользователей Python, то сможете направлять поисковые запросы или получать свежайшие извещения о новых программных продуктах, среди которых могут оказаться полезные Вам. Для регистрации в списке рассылки обращайтесь по адресу <http://www.python.org/psa/MailingLists.html>.

***Прим. В. Шипкова: готовьтесь к тому, что рассылка эта хоть и весьма полезна, но тем не менее — на английском языке!**

Резюме

В этой и предыдущей главах мы рассматривали в деталях процесс разработки модулей многократного использования. Был создан модуль `fixhash`, который мы усовершенствовали, протестировали и снабдили документацией. Вы узнали, что прежде чем приступить к разработке своих модулей, следует узнать, не сделал ли этого уже кто-нибудь до Вас. Вполне возможно также, что Вы найдёте готовые программные блоки, которые можно использовать в качестве компонентов Вашего модуля. Хотя мы и не рассмотрели подробно процесс поиска данных и программ в Internet, Вы узнали, где можно почерпнуть такую информацию.

Следующая глава завершает часть II. Мы создадим новый полнофункциональный класс и тщательно протестируем его работу. Впрочем, нашей конечной целью будет не создание класса, а усвоение на практике принципов объектно-ориентированного программирования. Вы увидите, что в следующей главе буду говорить не только я — программные коды будут говорить сами за себя.

Практикум

Вопросы и ответы

Насколько важно использование внешних модулей в современном программировании?

Без использования модулей Вы будете вынуждены каждый раз изобретать велосипед. В том, насколько важна и удобна возможность многократного использования модулей в программах, Вы очень быстро разберётесь, как только испытаете все преимущества на практике.

Контрольные вопросы

1. Как долго следует выполнять тестирование программы?
 - а) 25 раз.
 - б) 50 раз.
 - в) Пока она не будет успешно выполнена до конца хотя бы один раз.
 - г) Пока Вы не добьётесь сбоя программы, а затем заставите её успешно выполниться до конца.
2. Если Вы. нашли два модуля разных изготовителей, по какому принципу Вы выберете модуль для использования в своей программе?
 - а) Продукт той компании, которой Вы больше доверяете.
 - б) Тот модуль, который обладает большим быстродействием.
 - в) Тот модуль, который лучше задокументирован.
 - г) Тот модуль, который Вам рекомендовали друзья и знакомые.

Ответы

1. г. Программу следует трестировать до тех пор, пока Вам не удастся добиться сбоя в её выполнении. Не успокаивайтесь на том, что Вам удалось однажды успешно выполнить программу. Постарайтесь создать самые невероятные условия запуска программы до того, как этим займутся ваши пользователи.
2. в. Безусловно, все перечисленные факторы следует учитывать, но я бы всё же отдал предпочтение хорошо документированному модулю. Если модуль потенциально сильнее, но плохо описан, то Вы просто не сможете реализовать все его возможности, тем более, если у Вас нет опыта разбираться в хитросплетениях кода, написанного другим программистом.

Примеры и задания

Добавьте ещё одну опцию командной строки в модуль `testhash.py`, которая даст возможность пользователю включать или отключать показ сообщений об ошибках, устанавливая значение аргумента `verbose`. Рекомендую Вам поближе познакомиться с модулем `getargs`, который можно найти и загрузить по адресу <http://www.pauahtun.org/ftp.html>. Естественно, он хорошо задокументирован (<http://www.pauahtun.org/getargs.html>).

Если Вы захотите создать свою библиотеку модулей, Вам потребуется какой-нибудь архиватор файлов. Лучше всего подойдёт WinZip. Это приложение можно загрузить из Internet по адресу <http://www.winzip.com/>. Я предпочитаю использовать архиватор WinZip вместо `tar` или `gzip`, даже если предполагается использование архива в UNIX. Утилита `gzip` может распаковывать архивные файлы WinZip, а эти файлы меньше по размерам, чем родные файлы `gzip`.

***Прим. В. Шипкова: я бы рекомендовал к использованию WinRAR Александра Рошала.**

Если Вы больше хотите узнать о методах и приёмах тестирования программ, обратитесь к книге Brian Marrick, *The craft of software testing*, Prentice Hall, 1999 (Брайан Маррик, "Искусство тестирования программ"). Для Брайана характерен оригинальный подход к проблеме и живой стиль изложения. В такой же манере написана его статья *Classic Testing Mistakes* ("Классические ошибки при тестировании"), которую Вы можете найти по адресу <http://www.rstcorp.com/marick/classic/mistakes.html>. Внимательно просмотрите эту страницу. Здесь Вы найдёте ссылки на другие публикации по тестированию программ, например по адресу <http://www.rstcorp.com/marick>.

16-й час

Объекты в деле

В предыдущих двух главах мы сосредоточились на создании программных компонентов многоразового использования. В этой главе мы займёмся созданием сложного приложения, содержащего два класса и ряд функций, "склеивающих" компоненты в одно приложение. Основная задача для Вас состоит не в том, чтобы получить законченный продукт, который пригодится Вам в повседневной жизни, а в том, чтобы почувствовать себя уверенно при работе с кодом программы и научиться находить источники ошибок и средства их устранения. Мы последуем поговорке, которая гласит, что

лучше один раз увидеть, чем сто раз услышать. Это особенно верно для тех, кто хочет научиться программированию. Первое, чем должен в совершенстве овладеть программист, — это умением читать и понимать коды программ, написанные другими. Надо заметить, что чтение программ на языке Python, даже после того как в этой главе я намеренно попытаюсь всё запутать, несравнимо проще по сравнению с другими языками программирования. К концу главы Вас не испугает уже никакой программный код, каким бы длинным он ни был. Кроме того, Вы чётко уясните себе механизмы, с помощью которых программные блоки объединяются в единое приложение.

Создание приложения

Если Вы ещё помните, в главе 6 мы начали работу над программой учёта записей телепередач на кассетах видеомagnetofона. У меня скопилось более 5000 таких записей, поэтому я действительно остро нуждался в такой программе. Все кассеты были пронумерованы, как Вы помните, шестнадцатеричными цифрами. К проблеме учёта записей на кассетах мы ещё раз вернулись в главе 13, когда рассматривали концепции объектно-ориентированного программирования, а именно концепцию отношений между объектом-контейнером (кассетой) и объектом-членом контейнера (записью). Видеокассета, как и контейнер, может быть пустой, содержать несколько записей и быть переполненной — три возможных состояния класса. Действительно, кассеты и видеозаписи в нашем приложении будут реализованы с помощью двух соответствующих классов.

Свою коллекцию видеозаписей я начал давно, задолго до того, как познакомился с Python. Но уже в те времена не мог обойтись без помощи компьютера. Моя первая "база данных" представляла собой обычный текстовый файл с записями в определённом формате. Эта база данных считывалась утилитой на языке C, которая позволила мне автоматизировать выполнение многих рутинных задач. Впрочем, с её помощью нельзя было обновлять текстовые файлы, так что мне приходилось вносить изменения в базу данных с помощью обычного текстового редактора. Но программа на языке C сортировала записи, должным образом располагая их внутри кассет, затем выводила на печать списки записей в алфавитном порядке или по номерам, в зависимости от выбранной опции. Я добавил довольно сложные функции поиска записей, но затем понял, что впустую тратил время. Все необходимые операции поиска можно было выполнять быстрее и более эффективно с помощью утилиты `grep` (программа для UNIX, осуществляющая поиск образцов текста внутри текстовых

файлов). А после того как я перешел на Windows, мне потребовалась программа, отличающаяся лучшей переносимостью, чем мои ранее созданные утилиты на языке C. Лучшего выбора, чем Python, для этих целей трудно себе было даже представить.

Помимо того, что в программе, к рассмотрению которой мы сейчас перейдем, используются классы, привычные для программистов на языке C, мне удалось перенести почти без изменений ряд функций, ранее написанных на C. Некоторые дополнительные функции были разработаны уже в Python. Но в целом, как уже говорилось раньше, к представленной программе следует относиться не как к образцу профессионального дизайна (её можно было бы сделать и поэлегантнее), а как к примеру реализации различных подходов и концепций программирования для решения насущных практических задач. Если бы я начал работу над данной программой сейчас, она бы вышла значительно более объектно-ориентированной, а кроме того, значительно короче. Можете отнести к этому как к домашнему заданию.

Модуль shelve

В целом архитектура программы довольно проста. Видеокассеты содержатся внутри словаря, в котором шестнадцатеричные номера играют роль ключей. Каждая кассета, в свою очередь, содержит свой словарь записей, ключами которого выступают порядковые номера. На основе исходного текстового файла в оперативной памяти компьютера будет создаваться база данных. Чтобы затем сохранить эту базу данных на диске, воспользуемся модулем shelve. Сохранение баз данных таким путём называется *персистингом (persisting)*.

Модуль shelve позволяет сохранять любой объект в Python в двоичном формате и обрабатывать его затем как словарь. В обычных словарях в качестве ключей могут использоваться данные любых неизменяемых типов. Но в словаре, сохранённом на диске (иногда говорят — *положенном на полку*), ключами могут быть только строки. Больше информации о модуле shelve можно почерпнуть на Web-странице по адресу <http://www.python.org/doc/current/lib/module-shelve.html>. Я не стану описывать интерфейс этого модуля, но Вы во многом сможете разобраться во время анализа листингов далее в этой главе.

Формат записей для ввода

Чтобы максимально упростить процедуру ввода записей, в исходном текстовом файле был подобран формат, максимально

отвечающий требованиям языка C, т.е. строки текста можно было непосредственно сохранять в переменных языка C без предварительной обработки. Вот пример такой записи:

```
0001;0;sp,T120;89;89:
Masterpiece Theatre:
Last Place on Earth, The:
1: Poles Apart
```

Вся часть строки с самого начала до первого двоеточия представляла собой заголовок, в котором указывался номер кассеты, порядковый номер записи на кассете, скорость, размер записи, чистое и реальное время воспроизведения. За первым двоеточием следовали поля с описанием записи: категория (*Masterpiece Theatre*), название (*Last Place on Earth*), номер эпизода (1), название эпизода (*Poles Apart*). Поля были разделены комбинацией символов `:/t` (двоеточие с символом табуляции). Любая запись должна быть представлена хотя бы одним из этих полей. Кроме того, полезным может оказаться ещё ряд дополнительных полей. Например, для записей фильмов следует указать год, например (1999). Комментарии обычно заключают в фигурные скобки {}, а ключевые слова для поиска — в квадратные []. Кроме того, нужно предусмотреть возможность указать, что данная запись заменена более качественной или поздней версией. Ниже показан пример записи с дополнительными необязательными полями:

```
0045;2;ep,T120;139,139:
Hidden Fortress, The [Japanes] {Xd} (1958): ->119
```

Эта строка означает следующее: вторая запись на кассете № 45, фильм *The Hidden Fortress* ("Тайная крепость", Акиро Куросава (Akira Kurosawa)) японского производства 1958 г. К сожалению, окончание этого фильма было испорчено (символ Xd), поэтому новая версия была записана на кассету № 119.

В действительности формат записей не родился сразу, а постепенно дорабатывался мною. Так, например, за значением скорости `ep` раньше не было значения размера ленты `T120`. Вместо двух полей времени воспроизведения указывалось только реальное время записи на кассете, включая как фильм, так и сопровождающую его рекламу. В более позднем варианте уже указывались два значения времени: чистое время фильма и реальное время воспроизведения всей записи. Запись 59,46 означала, что воспроизведение всей записи занимает 59 минут, тогда как чистое время фильма — 46 минут, а остальное время идёт реклама. Нужно помнить, что все дополнения следует делать так, чтобы они были совместимы с

предыдущим форматом записей. Для нашего приложения нужно создать функцию, которая будет считывать строки исходного текстового файла, и для каждой новой кассеты создавать объект класса `cassette`, который должен "знать", как создавать, добавлять и удалять объекты записей телепередач. Функцию, обслуживающую ввод данных, мы рассмотрим позже. А пока сосредоточимся на конструировании двух классов кассет и записей.

Классы

В нашем приложении, назовем его `vcr` (video cassette records — записи на видеокассетах), используются два класса. Как мы уже знаем, это классы `cassette` и `program`. В листинге 16.1 показан класс `program` (нумерация строк соответствует положению этого класса в файле программы `vcr.py`).

Листинг 16.1. Класс `program`

```
class program
def __init__(self, s="",n=0):
    if s==" " and n==0:
        self.seq=-1
        self.year=None
        self.comments=None
        self.keywords=None
        self.rplcd=None
        self.tape=n
    else:
        self.year=None
        self.comments=None
        self.keywords=None
        self.rplcd=None
        self.tape=n

    i=string.index(s,":")
    h=header(s[0:i])
    self.seq=string.atoi(h[0])
    sp, tlen=parts(h[1],n)
    self.sp=tspeed(sp)
    if self.sp==-1:
        print "UNDEFINED Tape speed:",n,sp
        self.sp=1 # По умолчанию sp
    self.tlen=tsize(tlen)
    if self.tlen==-1:
        print "UNDEFINED Tape size:",n,tlen
    self.tlen=_tsizes["T120"]
```

```

        self.ttime, self.ctime=tuple(map(string.atoi,
list(parts(h[2],n)))
        stuff=s[i+2:]
        self.fields=[]
        q=string.splitfields(stuff, ":\t")
        for k in q:
            if alldigits(k):
                if k=="":
                    print h,"empty string",n
                    k=0
                self.fields.append(string.atoi(k))
            else:
                r=FindReplaced(k)*Возвращает[list,]
                if r!=None:
                    self.rplcd=r
            else:
                e=Extract(k)
                if e==None:
                    print "TAPE %04X Hosed."%(n)
                    raise ValueError
                if e[1]!=None:
                    self.year=e[1]
                if e[2]!=None:
                    self.comments=e[2]
                if e[3]!=None:
                    self.keywords=e[3]
                self.fields.append(e[0])

def __len__(self):
    return self.ttime

def speed(self):
    return speedname(self.sp)

def size(self):
    return sizename(self.tlen)

def __repr__(self):
    s="%02d: "%(self.seq)
    n=0
    for i in self.fields:
        if type(i)==type(0):
            s=s+'i'
        else:
            s=s+SwapThe(i)
        if n==0:
            if self.year!=None:
                s=s+" (%4d) *(self.year)

```

```

        if self.comments!=None:
            s=s+" {%s}"%(self.comments)
        if self.keywords!=None:
            s=s+" [%s]"%(self.keywords)
            s=s+', '
            n=n+1
    return s

```

Метод `__init__()`, наиболее длинный в этом классе, подразделён на два блока. Первый выполняется в случае, если функция `program()` вызывается без аргументов. Соответственно второй блок активизируется при передаче в класс вводной строки с номером кассеты. Без аргументов создаётся пустая заготовка объекта, в которую впоследствии можно внести информацию о записи. В случае передачи с аргументом вводной строки программа анализирует её содержимое и присваивает значения переменным-членам класса, представляющим все допустимые поля базы данных. Класс `program` содержит следующие переменные-члены:

- `seq` — порядковый номер записи на кассете;
- `year` — дата выпуска кинофильма (необязательное поле, используемое только для записей фильмов);
- `comments` — если исходная строка содержала комментарии, то они сохраняются в этой переменной;
- `keywords` — сюда добавляются ключевые слова, если они были выделены в исходной строке;
- `rp1cd` — в случае, если существует новая версия записи, здесь указывается её номер;
- `tape` — каждый объект видеозаписи знает номер своей кассеты;
- `sp` — скорость записи;
- `tlen` — размер ленты (Т120, Т30 и т.д.);
- `ttime` — чистое время передачи или фильма;
- `ctime` — время воспроизведения записи, включая время рекламы;
- `fields` — от 1 до `n` дополнительных полей, содержащих описание записи. Эти поля подразделяются на заголовки и числовые значения.

Комментарий: назначения большинства функций, вызываемых внутри метода `__init__()`, вполне очевидны или станут таковыми по мере дальнейшего рассмотрения кода программы. Единственная функция, с которой может возникнуть сложность, — это `string.splitfields()` (строка 315). Это то из немногих мест, где Python "стесняется" показать своё лицо. Данная функция принимает вводную строку (аргумент `stuff`) и разбивает её на элементы списка, используя в качестве границ комбинацию символов `':/t'`. Все поля исходной записи

теперь представлены соответствующими элементами списка. Поскольку комбинация символов-разделителей достаточно уникальна, данные в полях могут быть представлены практически любыми наборами символов. Содержимое полей анализируется далее. Например, выражение 'if alldigits(k)' (строка 317) определяет, содержит ли текущее поле только числовые значения, что будет соответствовать номеру эпизода, и т.д. Если название фильма состоит только из цифр, чтобы не спутать его с номером эпизода, перед названием следует установить обратную косую черту, например \2001.

Следующий класс `cassette` показан в листинге 16.2 (и вновь нумерация строк соответствует положению класса в файле `vcr.py`).

Листинг 16.2. Класс `cassette`

```
class cassette:
    def __init__(self, n=0, s=None):
        self.num=n
        self.l={}
        if s==None:
            pass
        else:
            t=program(s[5:], self.num)
            self.set(t)

    def key(self):
        return "%04X"%(self.num)

    def keys(self):
        if len(self.l)==0:
            return None
        k=self.l.keys()
        k.sort()
        return k

    def cat(self, s):
        if s!=None:
            t=program(s[5:], self.num)
            self.set(t)

    def __len__(self):
        t=0
        k=self.l.keys()
        k.sort()
        for l in k:
            i=self.l[l]
```

```

        if i!=None:
            t=t+i.tttime
        return t

def speed(self):
    s=tapespeed(self)
    return speedname(s[0])

def size(self):
    return tapesize(self)

def programs(self):
    return len(self.l)

def programlist(self):
    if isdummy(self):
        return None
    rslt=[]
    for i in self.l.keys():
        rslt.append(self.l[i])
    return rslt

def sequence(self, n):
    if type(n)==type(0):
        k="%02d"%(n)
    elif type(n)==type(""):
        k=n
    else:
        k="%s"%(n)
    if self.l.has_key(k):
        return self.l[k]
    return None

def set(self, p):
    try:
        n=p.seq
    except AttributeError:
        return
    k=self.sequence(n)
    if k==None:
        self.l["%02d"%(n)]=p
    else:
        self.l["%02d"%(n)]=p

def __repr__(self):
    s="%04X\n"%(self.num)
    if len(self.l)==0:
        s=s+"*No Tape*"

```



```

else:
    j=self.l.keys()
    j.sort()
    for k in j:
        i=self.l[k]
        s=s+'i'+"\n"
    return s

def nreplaced(self):
    if len(self.l)==0:
        return 0
    n=0
    k=self.keys()
    for i in k:
        p=self.l[i]
        if p.rplcd!=None:
            n=n+1
    return n

def getreplaced(self):
    if len(self.l)==0:
        return None
    d={}
    k=self.keys()
    for i in k:
        p=self.l[i]
        if p.rplcd!=None:
            d[i]=p.rplcd
    if len(d)==0:
        return None
    return d

def numeric(self, fid=0):
    x=self.programlist()
    return x

def alphabetic(self, fid=0):
    x=self.programlist()
    return x

```

Этот класс содержит только две переменные-члена:

- num — номер кассеты;
- l — словарь, содержащий объекты записей.

Да, так оно и есть. Записи телепередач в действительности хранятся в списке l. Функции нашего класса-контейнера сводятся не совсем к тому, о чем Вы могли подумать. Класс содержит только методы, ничего более сложного. Самый важный

среди них – метод `set()`, который помещает объект записи передачи в соответствующую позицию в списке `l`. Эта позиция определяется по порядковому номеру записи в кассете. Порядковый номер будет ключом в конечном словаре, причём, как Вы помните, чтобы сохранить этот словарь в файле на диске, ключ должен быть строкой. Процесс сохранения базы данных на диске в Python, между прочим, называют *pickling* (фаршировка). Поэтому не удивительно, что все функции, необходимые для сохранения файлов, хранятся в модуле `pickle`. А как же упомянутый выше модуль `shelve`? В действительности он является лишь интерфейсом модуля `pickle`, избавляя нас от необходимости разбираться в достаточно сложных функциях последнего.

Служебные функции

Для работы приложения нам необходим набор функций, которые будут обслуживать работу двух центральных классов. Сюда входят считывание исходных данных, построение базы данных в оперативной памяти компьютера, обработка строк, вводимых пользователем, и многое другое. На данном этапе мы пока не имеем ничего, кроме двух абстрактных объектов, которые сами по себе ничего не умеют делать. В листинге 16.3 показана оставшаяся часть кода модуля `vcr.py`. (Обратите внимание на разрывы в нумерации строк, где находятся блоки определений классов, которые мы рассмотрели раньше.)

Листинг 16.3. Служебные функции программы `vcr.py`

***Прим. В. Шипкова: здесь был длинный листинг описываемой программы. Я его убрал, зато положил ссылку на [файл-оригинал](#), если кому действительно надо. %)**

Назначение большинства этих служебных функций вполне очевидно. Кроме того, все функции снабжены документацией. Ниже приведены более подробные описания некоторых функций.

- `creatememdb()` – на основании строк исходного текстового файла создаёт в оперативной памяти компьютера базу данных. Адрес этой базы данных находится в системной переменной `VCRLIST`.
- `createfieldb()` – создаёт базу данных из оперативной памяти на жёстком диске.
- `openfiledb()` – создаёт и открывает файл базы данных для чтения и записи.
- `closedb()` – закрывает файл базы данных.

После создания файла базы данных он становится доступным для использования другими программами, в которые можно импортировать модуль `vcr` и использовать указанные выше функции для открытия и закрытия файла. Мы рассмотрим, как это делается, в следующих разделах.

Интерфейс командной строки

Поскольку программа на языке C была прообразом модуля `vcr.py`, нелишне посмотреть на распечатку опций командной строки этой программы, показанную в табл. 16.1.

Таблица 16.4. Опции командной строки программы `vcr` на языке C

Опция	Описание
<code>-h</code>	Вывод справки на <code>stderr</code>
<code>-?</code>	Вывод справки на <code>stdout</code>
<code>-V</code>	Напечатать версию и умереть
<code>-i<файл></code>	Ввести файл, если это не <code>'-/VcrList'</code>
<code>-o<файл></code>	Вывести файл (по умолчанию на <code>stdout</code>)
<code>-a</code>	Сортировать по заголовкам записей
<code>-r</code>	Обратный порядок записей
<code>-S<строка></code>	Выбрать по совпадению регулярного выражения
<code>-+</code>	Расширить область поиска-выделения дополнительными полями
<code>-K</code>	Ограничить область поиска-выделения только ключевыми словами
<code>-k</code>	Вывести ключевые слова на печать
<code>-E</code>	Не выводить описание
<code>-F<строка></code>	Выводить строку с использованием оператора форматирования <code>%</code>
<code>-H</code>	Справка по поводу форматирования выводимых строк
<code>-d<строка></code>	Отладка на уровне функций
<code>-c</code>	Отключение чувствительности к регистру во время поиска
<code>-N<строка></code>	Выбор кассет по номерам в диапазоне: <code>[x, x-x, -x, x-]</code>
<code>-T<строка></code>	Создание файла в формате HTML для выбранных записей
<code>-I<строка></code>	Создание указателя в формате HTML для выбранных записей
<code>-P</code>	Создание ярлыков PostScript для выбранных записей

-e	PostScript: только граничные ярлыки
-f	PostScript: только лицевые ярлыки
-R	Отчёт о суммарном времени воспроизведения всех лент
-p<строка>	Использование файла <code>-.pro</code> вместо заданных по умолчанию
-n##t	Установка ширины колонки в символах
-l	Выбор и показ самого длинного поля записей
-L	Исключительный выбор и показ самого длинного поля записей
-V	Отмена подстановки слов <code>a</code>, <code>an</code> и <code>the</code> в начале строки во время вывода её на печать
-O	Вывод файла в формате исходного файла

В нашем модуле базы данных `vcr` мы попытаемся воссоздать лишь часть описанных опций, хотя бы потому, что многие из них уже потеряли свой смысл или не существенны. Так, например, записи теперь хранятся не в сортируемом списке, как было в `C`, а в базе данных. Поэтому отпала необходимость в опциях управления списком. Определение аргументов командной строки лучше выполнить не в самом модуле `vcr`, а в отдельной программе, в которую предварительно импортируем модуль. Первая программа, которую мы создадим, будет просто посылать запрос в базу данных для нахождения в ней определённых кассет. Код этой программы показан в листинге 16.4.

Листинг 16.4. Программа `vcr.py`

```
#!C:\PYTHON\PYTHON.EXE

import os
import sys
import shelve
import string
import time
from vcr import cassette
from vcr import program
import vcr

try:
    _vcrfile=os.environ["VCRLIST"]
except:
    print "$VCRLIST not set in environment!"
    sys.exit(0)

if len(sys.argv)>1:
```

```

vdb=vcr.openfiledb()
for i in range(1, len(sys.argv)):
    t=vcr.findtape(sys.argv[i], vdb)
    if t!=None:
        print t, "Length:", len(t), "speed", t.speed(), \
            "size", t.size(), print "minutestleft", \
            vcr.minutesleft(t)
        print ""
k=t.keys()# Автосортировка
for j in k:
    y=t.sequence(j)
    # Функция принимает либо целые числа, либо строки.

    if y.fields[0]!=None:
        for z in y.fields:
            if type(z)!=type(0):
                print vcr.SwapThe(z)
            else:
                print z
        if y.year!=None:
            print "(%04d)" % ( y.year ),
        if y.comments!=None:
            print "{%s}" % ( y.comments ),
        if y.keywords!=None:
            print "[%s]" % ( y.keywords ),
print "%S~~~~~" % ( j )
vcr.closedb(vdb)

```

Обратите внимание, что мы не только импортировали в программу модуль vcr (инструкция `import vcr`), но также явно импортировали классы этого модуля (инструкция `from vcr import cassette` и аналогичная инструкция для класса `program`). Это необходимо было сделать, так как модулю `shelve` для оперирования файлами базы данных нужны имена исходных классов объектов. В программе предполагается, что аргументами командной строки могут быть только номера кассет, которые нужно отыскать в базе данных. Поиск производится с помощью функции `findtape()` нашего модуля `vcr`. В исходной версии на языке C допускался ввод диапазонов номеров в формате `1000-1FFF` или `-1FFF`. Я планировал добавить такую возможность поиска в модуль `vcr`, но не успел. Может быть, Вы это сделаете сами.

Следующая программа, также работающая в режиме командной строки, просматривает все кассеты в базе данных и выбирает те из них, на которых свободного места осталось больше, чем указано в аргументе. Такая утилита будет весьма полезной, например, когда Вам нужно записать новую 1-часовую

программу и Вы ищете подходящую для этого кассету. Код такой программы показан в листинге 16.5.

Листинг 16.5. Программа vcrleft.py

```
#!/usr/local/bin/python

import os
import sys
import shelve
import string
from vcr import cassette
from vcr import program
import vcr

try :
    _vcrfile = os.environ[ "VCRLIST" ]
except :
    print "$VCRLIST not set in environment!"
    sys.exit(0)

spd=0

howmuch=28.0
lessthan=9999.0
tlist=[]
mlist=[]

def prlist(list):
    tpr=0
    n=len(list)
    z=0
    i=0
    for z in range(n):
        print list[z],
        tpr=tpr+1
        i=z%3
        if i==2:
            print ""
        else:
            print "\t",
        if i!=2:
            print ""
        print "-----"
    print "-----"

if len(sys.argv)>1:
    howmuch=float(string.atof(sys.argv[1]))
if len(sys.argv)>2:
```

```

try:
    lessthan=float(string.atoi(sys.argv[2]))
except :
    if sys.argv[2]=="sp":
        spd=1
    elif sys.argv[2]=="lp":
        spd=2
    elif sys.argv[2]=="ep" or sys.argv[2]=="slp":
        spd=3
    else:
        print "I don't understand", sys.argv[ 2 ]
        raise ValueError

if len(sys.argv)>3:
    if sys.argv[3]=="sp":
        spd=1
    elif sys.argv[3]=="lp":
        spd=2
    elif sys.argv[3]=="ep" or sys.argv[3]=="slp":
        spd=3
    else:
        print "I don't understand", sys.argv[3]
        raise ValueError
vdb=vcr.openfiledb()
lst=vcr.getkeys(vdb)
lst.sort()
print "Looking for", howmuch, "<", lessthan, "minutes, in",
len ( lst ), "Keys"
print "-----"
print "-----"
for j in lst :
    i=vcr.findtape(j, vdb)
    t=vcr.minutesleft(i)
    if spd!=0:
        if not vcr.onlyspeed(i, spd):
            continue
    if howmuch<t<lessthan:
        x, y=vcr.tapespeed(i)
        hl, ml=vcr.tupleize(t)

st="T%04X%s -- %03dm %02d:%02d"%(i.num, vcr.speedname(x ),
t, hl, ml)
mlist.append(st)
st="%03dm %02d:%02d -- T%04X %s"%( t, hl, ml, i.num,
vcr.speedname(x) )
tlist.append(st)

tlist.sort()
mlist.sort()

```

```
prlist(mlist)
prlist(tlist)

vcr.closedb(vdb)
```

Большая часть кода этой программы самоочевидна. Программа ожидает передачи с аргументом командной строки значения времени в минутах, необходимого для новой записи, а также значение скорости записи. Если параметр времени пропущен, а передано только значение скорости, то по умолчанию переменной `howmuch`, принимающей искомое время, присваивается значение 28 минут.

Резюме

В этой главе, последней в части II, мы впервые столкнулись с длинными кодами программ, подразделенными на модули, классы и служебные функции. Некоторые части кода, например, там, где используются регулярные выражения, остались для Вас непонятными. Но Вы можете разобраться в них самостоятельно, почерпнув дополнительную информацию из источников, рекомендованных ниже, в разделе "Примеры и задания". Целью, которую я ставил перед собой, выбирая программный код для демонстрации, было наглядно показать Вам, как взаимодействуют разные части программы. Все рассмотренные программы можно загрузить с Web-страницы этой книги. Там же Вы найдёте небольшой подготовленный мной файл базы данных в текстовом формате. Только предупреждаю, это всего лишь демонстрационная база данных, не просите меня в письмах одолжить ту или иную кассету для просмотра.

***Прим. В. Шипкова: в приведённой базе данных, есть как ковбойские вестерны, так и классическое японское кино. 80% этих фильмов я не видел, ничего существенного добавить не могу.**

Практикум

Вопросы и ответы

Всё, что мы делали, изучая материал данной главы, — это рассматривали длиннющие программные коды. Надо ли нам теперь садиться за испытание этих программ?

Не обязательно. Вес, что Вам сейчас надо, — это немножко отдохнуть перед следующей частью.

Примеры и задания

Лучшая книга, описывающая использование регулярных выражений, написана Jeffrey E. F. Friedl, *Mastering Regular Expressions*, O'Reilly & Associates. Давид Ашер (David Ascher) так сказал о ней: "Книга Джеффри Фриедла одна из немногих, заслуживающих ту цену, которую я за неё заплатил, даже при том, что представления Джеффри о Python несколько устарели, а я довольно редко использую на практике регулярные выражения. Это лишний раз доказывает стремление человеческого сознания познать непознанное, даже если оно не нужно". От себя добавлю, если Вам всё же нужны регулярные выражения, то лучшего источника данных о них Вы не найдёте.

Сделайте возможным поиск кассет в базе данных vcr по указанным диапазонам. Подсказка: воспользуйтесь в качестве образца модулем `getargs`, который можно найти и загрузить по адресу <http://www.pauahtun.org/ftp.html>. Не сомневайтесь, этот модуль хорошо документирован (<http://www.pauahtun.org/getargs.html>).

Часть III

Графический пользовательский интерфейс приложений на языке Python

Темы занятий

17. Введение в графический пользовательский интерфейс Python
18. Графические объекты библиотеки Tk, I
19. Графические объекты библиотеки Tk, II
20. Функции рисования библиотеки Tk, I
21. Функции рисования библиотеки Tk, II
22. Функции рисования библиотеки Tk, III
23. Наборы Мандельброта
24. Прочие средства

17-й час

Введение в графический пользовательский интерфейс Python

С этой главы мы начнём знакомство с библиотекой Tkinter, представляющей средства разработки графического пользовательского интерфейса (GUI – Grafical User Interface) в Python. Изучению этой библиотеки будет посвящена почти вся завершающая часть данной книги. Вы увидите, что принципы программирования приложения с

графическим пользовательским интерфейсом существенно отличаются от принципов создания программ для работы в режиме командной строки. Поэтому мы начнём занятие с рассмотрения разных моделей программирования и теоретических основ (не беспокойтесь, эти теории не такие уж и сложные). Закончив чтение главы, Вы сможете описать отличия программирования для командной строки и GUI; объяснить, что такое очередь событий и что вообще в программировании понимают под термином события; объяснить, что такое функции обработки событий.

***Прим. В. Шипкова: в вопросах управления окошками, Python хоть и является интерпретатором, но сделан куда более грамотно, чем скажем Visual Basic.**

Это ещё одна из тех скучных глав, где мы практически не будем работать на компьютере, а будем только постигать абстрактные идеи. Но ничего, при изучении материала следующих глав Вы сможете вволю поупражняться за компьютером.

Модель программирования для GUI

***Прим. В. Шипкова: GUI - graphical user interfase - графический интерфейс пользователя. Собственно, само окошко с кнопками и рюшечками. GUI - общепринятое сокращение.**

Я помню, как появился компьютер Lisa компании Apple. Вице-президент компании, в которой я тогда работал, как раз приобрел такой компьютер. Он установил его в общем компьютерном зале, где с ним могли поработать все желающие. Надо сказать, что в этой компании даже секретарши были профессиональными компьютерщиками, поэтому можете себе представить, сколько часов этот компьютер наработал. Безусловно, что и я не мог отказать себе в удовольствии посидеть за этим компьютером и получил массу удовольствия от его интерфейса. Несколько месяцев спустя возникла компания Macintosh, и многие инженеры, ранее просиживавшие за компьютером Lisa, приобрели себе машины этой компании. В то время я не мог себе позволить купить Macintosh, поэтому мой опыт общения с Mac GUI сводился к работе с принтером PostScript, подключенным к единственному компьютеру Mac, приобретенному нашей компанией для общего пользования. Чтобы изучить работу с PostScript, в то лето мне приходилось подолгу задерживаться каждый вечер.

***Прим. В. Шипкова: PostScript - скриптовый язык управления принтерами. Питон также является скриптовым языком. Только**



для всего. ;) На сегодняшний день наибольшим распространением известен PostScript3.

Много позже, когда уже появилась X Windows, я для работы над одним проектом получил рабочую станцию Sun, которую умиленно называл про себя "спасателем", хотя мои сотрудники упорно называли её "рыболовным крючком". Это было связано с тем, что незадолго до этого у нас в компании возникли серьезные проблемы с программным обеспечением сервера времени.

***Прим. В. Шипкова: X Window – графический сервер (т. е. специальная фоновая программа) для *nix (Unix, Linux, FreeBSD, OpenBSD, Sun Solaris). В Windows графика встроена в ядро, что на самом деле не такая уж и хорошая идея (хотя незначительный прирост скорости в графических приложениях отмечается) .**

Программное обеспечение сервера времени предназначено для поддержания единого системного времени для всех рабочих станций, подключенных к сети. Наша программа хорошо работала в течение нескольких часов, после чего вдруг возникал провал во времени с отставанием на час, а то и больше. В конце концов, нам удалось обнаружить ошибку в коде. Она состояла в том, что в одной из ветвей инструкции `if`, где предполагалось добавление значения времени, происходило вычитание. Надо сказать, что прежде чем обнаружить ошибку, данный код просматривало как минимум 4 человека, причём по несколько раз, пока один из них не пожертвовал своим временем и не выполнил отладку программы в пошаговом режиме. Только тогда удалось обнаружить, что значение времени, вместо того чтобы прирасти, уменьшилось. Хотя отладка программ на языке Python выполняется гораздо проще, чем при использовании других языков программирования, и мы даже не рассматриваем эту тему в данной книге, запомните эту историю. Часто только выполнение программы в пошаговом режиме (а интерпретатор Python предоставляет эту возможность) позволяет обнаружить недопустимые изменения значений переменных, что и приводит к сбою программы, возможно, даже в другой части кода. Четыре человека, включая меня, пренебрегли этим средством, в результате чего наша компания в течение месяца работала в стрессовом режиме.

Моя первая программа с интерфейсом GUI подключалась по сети к серверу Морской обсерватории Соединенных Штатов (U.S. Naval Observatory), определяла текущее время, устанавливала системное время на моём компьютере Sun, после

чего по этому образцу устанавливала время на всех компьютерах корпоративной сети в обход сбойного программного обеспечения сервера времени. Вот почему свой компьютер я называл спасателем.

Таким образом, свой первый опыт обращения с графическим пользовательским интерфейсом я получил при работе с X Windows на компьютере Sun (по-моему, это была версия программы X11 2-го выпуска или последний выпуск версии X10). Работа с графикой оказалась не особенно сложной, но то, над чем мне действительно пришлось поломать голову, так это над парадигмой обработки событий, на чём основана работа любого приложения с GUI. Во всех программах, которые мы написали до сих пор, в основу были положены средства управления логикой выполнения программы. Пользователь запускал программу, после чего программа, основываясь на исходных данных, говорила пользователю, что нужно делать. После выполнения программа автоматически завершалась.

Программы с интерфейсом GUI также запускаются пользователем. В ответ они выводят на экран графические объекты, такие как диалоговые окна или окна текстового редактора. После этого выполнение программы приостанавливается в ожидании дальнейших команд пользователя. Программа завершается только после того, как пользователь даст соответствующую команду.

Логика работы программы с командной строкой обычно достаточно прямолинейна. После запуска такая программа оказывается в пункте А, после чего переходит к пункту В, затем к пункту С и т.д. до пункта Z, где происходит её закрытие. Все это записано в ядре программы: начать здесь, перейти туда, завершиться там. Программы с интерфейсом GUI тоже имеют этап инициализации, но ядро программы создаётся совершенно иначе. Как правило, это очень простая модель, которую можно выразить следующим псевдокодом:

```
while событие:  
    .....  
do функции обработки события
```

Управление программой основано на анализе событий, а не на инструкциях логики выполнения программы. Может показаться, что это небольшое отличие. В действительности отличие фундаментальное, и скоро Вы в этом убедитесь. Программы командной строки проходят жёстко определённую последовательность событий: А, В, С, ..., Z. Программы с интерфейсом GUI автоматически выполняют только этап инициализации, после чего ожидают появления событий,

задаваемых пользователем. Поскольку последовательность событий в данном случае зависит только от пользователя, порядок выполнения программы непредсказуем. И наконец, программы GUI завершают свою работу только по команде пользователя. Можно сказать, что основное отличие лежит в предсказуемости выполнения программы. Выполнение программ командной строки жёстко детерминировано (по крайней мере в теории) и зависит только от исходного набора аргументов.

***Прим. В. Шипкова: строго говоря, и в командной строке, при работе с вариантами действий, например:**

- 1. Повторить**
- 2. Выход**

реально невозможно предсказать какой пункт выберет пользователь, и уж тем более, какую клавишу он нажмёт. [Такие ушлые пользователи часто являются головной болью программиста] ;) [Впрочем, такие профессиональные ушлые пользователи называются тестеры, им даже иногда приплачивают] ;)

Выполнение программ GUI непредсказуемо не в смысле возвращения правильного результата, а в том смысле, что даже при запуске программы в идентичных исходных условиях невозможно предсказать, какие команды и в какой последовательности будет давать ей пользователь.

Как обрабатываются события? Это зависит от системы и конкретного программного обеспечения. Так, в X Windows события перехватываются сервером и передаются отдельным программам-клиентам. Оба приложения, как сервер, так и клиент, в этом случае относятся к так называемому *пользовательскому уровню*, т.е. они не являются частью операционной системы, а запускаются пользователями. Во всех версиях Windows показ окон интегрирован с самой операционной системой. Программа X Windows не относилась к системе и запускалась как обычная программа поверх операционной системы, как правило, UNIX. Операционная система Macintosh, по крайней мере те ранние версии, с которыми я работал, также содержала встроенную систему показа окон и не имела интерфейса командной строки. Но, насколько я знаю, в более поздних версиях, таких как MacOS X, операционная система и графический интерфейс пользователя были отделены друг от друга. Хотя, честно говоря, я не самый большой специалист по Macintosh.

***Прим. В. Шипкова: последние версии MacOS X это не иначе, как доработанная FreeBSD. Я бы очень хотел поработать с этой портированной системой на x86 - Apple'вцы обещали...**



В любом случае, события генерируются пользователями самыми различными способами: перемещением мыши, нажатием клавиш клавиатуры или щелчком кнопкой мыши. Сгенерированные события помещаются в очередь, откуда извлекаются сервером (не важно, является ли он при этом частью операционной системы или нет). Сервер помещает события в другую очередь, из которой в этот раз их извлекают приложения пользователей. Теперь важно понять, что собой представляет очередь событий.

Вы можете представить себе очередь событий как трубку, в которую помещаются сухие горошины (или шарики, если хотите). После того как горошина помещена в трубку, извлечь её можно только с другого конца. Горошина недоступна, пока она находится внутри трубки. Но после того как Вы извлекли горошину, Вы можете делать с ней всё, что угодно, например сварить суп или поместить в другую трубку-очередь. Программа или сервер может поместить событие в очередь, и она никуда не денется оттуда до тех пор, пока сервер или программа не извлекут его с другого конца очереди для обработки. Именно это и показано в нашем псевдокоде с циклом `while`, представленном выше. Это бесконечный цикл, который извлекает событие из очереди, обрабатывает его и возвращается в исходное состояние в ожидании следующего события.

Существуют две модели обработки событий: *синхронная* и *асинхронная*. Другими словами, в теле `do` нашего цикла `while` функции обработки событий могут вызываться синхронно или асинхронно. В первом случае выполнение тела цикла будет приостанавливаться после каждого вызова функции обработки события до возвращения ею какого-либо результата. Никаких других операций, кроме самой функции, в это время выполняться не будет. В асинхронной модели вызов функции происходит в теле `do`, после чего, не ожидая завершения вызванной функции, цикл продолжается извлечением следующего события в очереди и его обработкой. В результате в одно и то же время может выполняться несколько функций обработки событий. Программы с интерфейсом командной строки, как правило, выполняются синхронно.

В UNIX программы с интерфейсом командной строки и асинхронным выполнением называются демонами (daemon). Аналогичные программы в Windows называются службами. Программа поддержки сервера времени, о которой я рассказывал в начале главы, как раз была демоном, запускаемым в UNIX. Соответствующие службы сервера времени доступны в Windows NT. Если разобраться в структуре



программного кода демона или службы, то мы увидим всё тот же цикл `while`, ожидающий поступления событий для обработки.

***Прим. В. Шипкова: фоновый процесс, выполняющие сервисные функции, я бы предпочёл, всё-таки называть "службой". "Демон" не предполагает действий в интересах системы, и уж тем более по отношению к пользователю. Отдельных людей демоны даже замуровывали. ;)**

Оконные системы, основанные на циклах обработки событий, практически всегда являются асинхронными. Чтобы поддерживать необходимый уровень интерактивности взаимодействия с пользователем, такая система не может позволить себе замирать в ожидании выполнения каждой вызванной функции (хорошо, если выполнение очередной функции вообще когда-нибудь завершится), вместо того чтобы отслеживать текущие перемещения мыши по коврику и нажатия клавиш. Кроме того, многие события протекают в фоновом режиме, оставаясь незаметными для пользователя. К ним можно отнести изменения положений окон на экране, обновление окон и файлов, таких как файлы `.ini` в системе Windows. Некоторые события вызывают за собой целые цепочки других событий (в терминологии Windows – сообщений), каждое из которых нужно отследить и обработать.

Вам уже встречались такие термины, как выполнение процедур в фоновом режиме и на переднем плане. Эта терминология уходит корнями к самым первым мультипроцессорным компьютерам. В компьютере IBM 360 память подразделялась на две части: переднего плана и фоновую, которые назывались областями памяти. Чтобы программа работала в фоновом режиме, её нужно было особым образом скомпилировать. Фоновые программы выполнялись значительно медленнее, чем программы переднего плана, но и выполнение последних при этом существенно замедлялось. На первых этапах это было допустимо, но с точки зрения современных стандартов такой подход просто невозможен. Стоит напомнить, что самый мощный компьютер IBM 360 имел 256 Кбайт памяти (обратите внимание – килобайт, а не мегабайт), тогда как на моём компьютере сейчас установлено 320 Мбайт ОЗУ.

Системы Windows 95/98 и Windows NT выполняют в фоновом режиме колоссальную работу по постоянному обновлению системного реестра, который подменил собой большинство файлов `.ini`. Впрочем, принцип остался тот же: все текущие изменения установок и параметров работы программ заносятся в соответствующие папки реестра все время, пока Вы работаете с компьютером. В отличие от X Windows, которая обслуживала только себя, оставив все системные проблемы на

UNIX, системе Windows ещё приходится поддерживать работу клавиатуры, дисплея, жесткого диска и других присоединенных устройств. Ни одна из этих задач не может ожидать, пока Вы доиграете свою партию в "Солитер".

События

Вы уже узнали, как события генерируются и передаются с использованием особой структуры данных — очереди событий. Теперь Вам нужно выяснить, какими бывают события и ответы на них. В следующем разделе мы рассмотрим способы обработки различных событий.

Все оконные системы поддерживают взаимодействие монитора с клавиатурой и мышью. Монитор (обычно) является пассивным устройством вывода для показа окон, заставок, часов, панелей задач и других графических объектов. Клавиатура и мышь — это активные устройства. Можно нажать клавишу на клавиатуре, переместить мышь по коврику и щелкнуть её кнопкой. Каждое подобное действие генерирует событие или даже несколько событий. Например, с клавишей клавиатуры связаны события "нажать и отпустить", а также нажатие клавиши в сочетании с нажатой управляющей клавишей <Shift> или <Ctrl>. Изменение положения указателя мыши на экране отслеживается X и Y, где X указывает смещение по горизонтали, а Y — по вертикали от верхнего левого угла. Какое событие будет вызвано щелчком правой или левой кнопкой мыши, зависит от текущего положения курсора, а также от того, удерживались ли¹ в это время нажатыми управляющие клавиши <Shift>, <Ctrl> или <Alt>.

Поскольку нас интересуют лишь те события, которые можно обрабатывать с помощью Python и функций библиотеки Tkinter, мы не станем рассматривать многочисленные события (их несколько сотен), определённые в системе Windows. Большинство из них недоступны для Python. Если всё же возникнет необходимость в их использовании, следует использовать модуль расширений Марка Хаммонда (Mark Hammond) Win32. Чтобы узнать больше об этом модуле и загрузить его на свой компьютер, посетите узел по адресу <http://starship.python.net/crew/mhairanond/>.

В табл. 17.1 перечислены события, используемые в Tkinter.

Таблица 17.1. События, поддерживаемые в Tkinter

Событие	Описание
Activate	Данное окно или графический объект становятся активными

ButtonPress, Button	Нажатие первой, второй кнопки мыши, соответственно Button1 и т.д., Double-Button- 1 –двойной щелчок первой кнопкой мыши. Можно даже задать тройной щелчок: Triple-Button-1
ButtonRelease	Отпускание кнопки мыши
Circulate	Изменение позиции окна или графического объекта в Z-последовательности
Colormap	Изменение цветовой схемы. (По-моему, это событие не происходит в Windows.)
Configure	Изменение размера, формы или положения графического объекта
Deactivate	Данное окно или графический объект перестают быть активными
Destroy	Данный графический объект назначен для удаления
Enter	Указатель мыши пересек границы данного окна или графического объекта (оказался над объектом)
Expose	Некоторые или все окна и графические объекты изменили экспозицию
FocusIn	Данный графический объект получил фокус
FocusOut	Данный графический объект потерял фокус
Gravity	Смысл этого события трудно объяснить начинающему программисту
KeyPress, Key	Нажатие клавиши; Keya – нажатие клавиши a и т.д.
KeyRelease	Отпускание клавиши
Motion	Перемещение указателя мыши
Map Property	Данное окно или графический объект были выведены на экран
Reparent	Некоторое свойство (например, цвет) окна или графического объекта было изменено
Unmap	Родительским для данного окна/графического объекта стало другое окно/графический объект
Visibility	Данное окно или графический объект не были выведены на экран Изменился вид отображения окна: свернуто, восстановлено и т.д.

Leave

Указатель мыши вышел за границы данного окна или графического объекта

Как видите, набор контролируемых событий достаточно обширный. Но многие из этих событий используются довольно редко. И уж тем более можно утверждать, что в одной программе Вы не будете использовать все события. По мере изучения материала этой части смысл многих событий станет для Вас более понятным. Чтобы программа реагировала на события, нужно зарегистрировать его и написать выполнение функций обработки событий. Если какое-либо событие не будет зарегистрировано в программе, то такое событие пройдет для программы незамеченным. Регистрация и функции обработки событий будут рассмотрены в следующем разделе.

функции обработки событий

Чтобы зарегистрировать событие, сначала нужно создать соответствующий *графический объект* (в английской терминологии – *widget*).

***Прим. В. Шипкова: читается, как виджет (визуальный компонент). Также известно слово "гаджет". Это не от слова "гадить". ;) Гаджетом называют маленькую вещь, выполняющую какую-либо не слабую функцию, например, mp3-плеер.**

Под графическими объектами понимают окна, снабжённые специальными свойствами. Каждый раз, когда запускается программа с графическим пользовательским интерфейсом, на экране появляются связанные с ней графические объекты, ответственные за выполнение различных функций программы. Примерами графических объектов могут быть кнопки, текстовые поля, меню, полосы прокрутки и ярлыки. Окна, которые содержат другие окна меньшего размера, называются *родительскими графическими объектами*. Родительским для всех окон и графических объектов вашей программы является *корневое окно*. Все графические объекты имеют свой *родительский объект*.

Когда Вы запускаете редактор IDLE, на экране появляется множество графических объектов, например строка меню и раскрывающиеся меню. Программа IDLE полностью написана на Python с использованием компонентов Tkinter. Прежде чем создать свой первый графический объект, Вы должны выполнить операцию инициализации:

```
from Tkinter import *  
root=Tk()
```

Эти строки импортируют модуль Tkinter и инициализируют переменную `root` объектом корневого окна. После того как Вы получили в программе объект корневого окна, можно приступить к созданию графических объектов:

```
button = Button(root)
```

Результат выполнения этого выражения самоочевиден, не так ли? Вы создали объект кнопки `button`, которая, впрочем, пока что ничего не делает и даже не видна на экране. Следовательно, нам нужно ещё немного поработать над ней. Прежде всего добавим надпись, которая будет отображаться на кнопке при её показе на экране:

```
button["text"] = "Завершить работу"
```

В данном выражении объект кнопки `button` обрабатывается как словарь. (В действительности это не совсем так, но мы можем представить себе, что кнопка — это словарь, тем более что объект кнопки содержит метод `__getattr__()`, с которым мы познакомились в главе 13.) Интерпретатор находит в объекте кнопки атрибут `"text"` и присваивает ему значение `"Завершить работу"`. С этого момента мы видим кнопку на экране, хотя с ней всё ещё не будет связано выполнение никаких функций. Чтобы придать кнопке функциональность, нужно зарегистрировать для неё событие и написать функцию обработки этого события. Вот как это делается:

```
def die(event):  
    sys.exit(0)  
  
button.bind("<Button1>", die)
```

Вот, собственно, и всё! Мы определили функцию `die()`, затем с помощью метода `button.bind()` назначили её функцией обработки события щелчка мыши. Метод `bind()` принимает два параметра. Первый строковый параметр задаёт имя события, которому Вы хотите назначить функцию обработки. Треугольные скобки `<` и `>` необходимы в тех случаях, если имя события не соответствует в точности именам, показанным в первом столбце табл. 17.1. Второй параметр указывает имя функции обработки события, которая может принимать единственный параметр — `event` (событие). Наша функция `die()` не нуждается ни в каких параметрах, но ниже мы рассмотрим ситуации, когда параметр необходим.

Есть альтернативный способ определения функции обработки события. Многие графические объекты содержат специальные атрибуты, с которыми напрямую можно связывать функции

обработки событий. В случае с кнопкой альтернативный код будет выглядеть так:

```
def die():  
    sys.exit(0)  
  
button["command"]=die
```

При таком подходе параметр `event` в функции обработки события никогда не используется.

После того как мы назначили любым из методов функцию обработки события, для получения функционирующей программы нам осталось выполнить ещё несколько операций. Прежде всего нужно добавить следующие две строки:

```
button.pack()  
root.mainloop()
```

Метод `pack()` называют менеджером геометрии объекта. Он посылает объекту кнопки сообщение, определяющее её размер таким образом, чтобы она могла вместить назначенную ей надпись. Метод `mainloop()`, вызванный для корневого окна, говорит примерно следующее: "ОК, я завершил работу по своей настройке и инициализации и теперь готов к приему и обработке событий". Говоря формальным языком, с помощью этого метода мы создали цикл обработки событий, соответствующий рассмотренному ранее псевдокоду бесконечного цикла с оператором `while`.

Введите указанные выше строки и сохраните код в файле, например, под именем `btn.py`. Код полученной программы в полном виде показан в листинге 17.1.

Листинг 17.1. Программа `btn.py`

```
#!C:\PYTHON\PYTHON.EXE  
  
import sys  
from Tkinter import *  
  
def die(event):  
    sys.exit(0)  
  
root=Tk()  
button=Button(root)  
button["text"]="Quit"  
button.bind("<Button-1>", die)
```

```
button.pack()  
root.mainloop()
```

Теперь запустите программу как обычно. Результат показан на рис. 17.1.

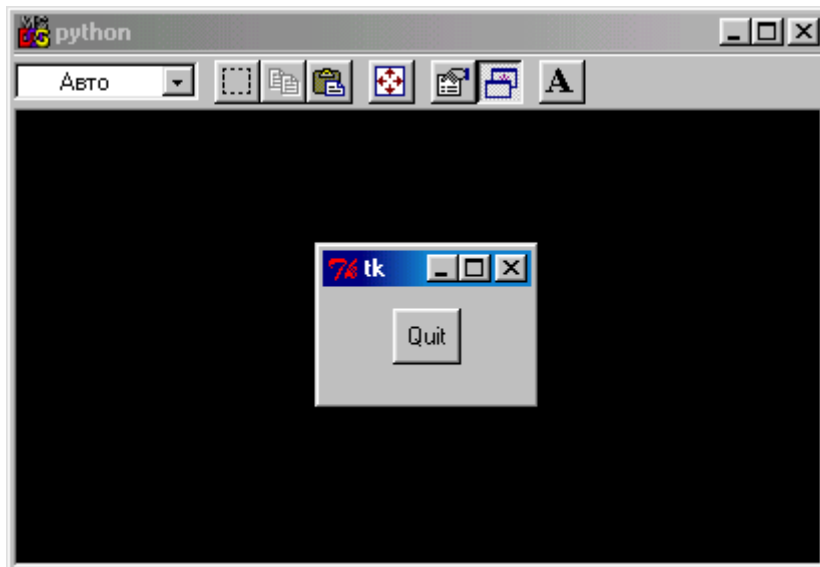


Рис. 17.1. Выполнение программы *btn.py*

*Прим. В. Шипкова: на самом деле кнопка будет выглядеть иначе. Это связано с изменившейся библиотекой Tk. Кроме того, если Вы захотите использовать русские надписи, то придётся или использовать кодировку utf8, или писать примерно следующее:

```
button["text"]=u"Выход"
```

Символ 'u' говорит о том, что строку следует преобразовать в кодировку Юникод(Unicode).

Как видите, функции обработки событий в Python мало чем отличаются от обычных функций. В языках C и C++ функции обработки событий значительно сложнее. Это касается как их регистрации, так и выполнения. Ниже показан пример такой функции в языке C.

```
static  
void ButtonCallback(Widget W, XtPointer cd, XtPointer dd)  
{  
    /* Процедура завершения программы. */  
    exit(0);  
}
```

А чтобы зарегистрировать событие, в языке C нужно ввести следующую строку:

```
XtAddCallback(w, XmNactivateCallback,  
ButtonCallback, (XtPointer)cd);
```

Естественно, что выше приведён простейший пример. Обычно код функций обработки событий занимает много строк. Кроме того, потребуется ещё много вспомогательных программных блоков, которые не нужны в Python. Это строки инициализации, определения графического объекта и настройки его свойств. А затем ещё нужно будет самостоятельно написать цикл обработки событий, который, впрочем, можно будет реализовать как простой цикл while. Наконец, Вам нужно скомпилировать полученную программу, прежде чем Вы сможете ею воспользоваться. Напротив, Python, имеющий модуль Tkinter, позволяет выполнить всю необходимую работу в считанные минуты, и перед Вами на экране уже готовый к употреблению полнофункциональный графический объект. Программа на языке C для запуска в X Windows, аналогичная программе btn.py, заняла бы как минимум сотню строк кода, в котором начинающий допустил бы сотню ошибок.

***Прим. В. Шипкова: поверьте мне на слово – сотня ошибок на сотню строк в Си – это не метафора. :)))**

Резюме

Мы немного разобрались в принципах создания графического пользовательского интерфейса и в том, чем программы GUI принципиально отличаются от программ командной строки. Кроме того, Вы узнали, что представляют собой события, очереди событий и функции обработки событий. В следующей главе мы подробно познакомимся с графическими объектами, предоставляемыми библиотекой Tkinter.

Практикум

Вопросы и ответы

Вы же говорили, что мы не будем писать на этом занятии никаких программ?

Да, я так думал, но не удержался.

Каким был самый мощный из всех компьютеров модели IBM 360?

Мне приходилось работать на модели IBM 360/90 с 1 Мбайт оперативной памяти. Насколько я помню, в то время это был крупнейший компьютер во всем округе Вашингтон (Washington DC). Он был настолько большим, что занимал огромный зал, а

во время работы выделял столько тепла, что его пришлось снабдить системой водяного охлаждения.

Контрольные вопросы

1. Какая метафора лучше всего подойдёт для описания работы очереди событий?
 - а) Это как стопка тарелок в кафетерии – тарелку, которую последней положил, первой и взял.
 - б) Это как верёвка – тянешь за один конец, а на другом конце кто-то орет.
 - в) Это как трубка, в которую опускаешь шарик, а на другом конце вынимаешь.
 - г) Это как эскалатор – встал и поехал.
2. Кто определяет, когда должна завершиться программа с интерфейсом GUI?
 - а) Программист. Программа завершается, когда сделает всю работу.
 - б) Компьютер. Как только программа исчерпает всю доступную память, её нужно закрыть.
 - в) Операционная система. Как только программа ей надоест, она покажет синий экран смерти.
 - г) Пользователь. Когда он завершит работу с программой, то даст команду на закрытие.
3. Что такое функция обработки событий?
 - а) Функция или метод, которые возвращают события.
 - б) Зарегистрированные метод или функция, которые вызываются в ответ на указанное событие.
 - в) Метод или функция, которые вызывают другую функцию, назначенную данному событию.
 - г) Часть операционной системы Windows, разработанная для обработки стандартных событий.

Ответы

1. в. Лучшей метафорой для описания работы очереди будет трубка, заполненная шариками. Стопка тарелок – это традиционная метафора для описания работы стека (ещё одной стандартной структуры данных). Очередь поддерживает исходный порядок событий. Если событие А предшествовало событию В, то событие А будет первым извлечено из очереди.
2. г. Когда должна завершиться работа программы с GUI, решает только пользователь. Этим программы GUI

отличаются от программ командной строки, которые завершаются автоматически после выполнения всего кода программы.

***Прим. В. Шипкова: это справедливо, но не совсем. Если пользователь знает, что выполнение операции займёт много времени, например сжатие фильма, то он может поставить галочку "по окончании работы выключить компьютер" (при условии, что такая галочка в программе есть). Тогда не то что программа произвольно закроется, компьютер выключится!**

3. б. функция обработки события - это зарегистрированные метод или функция, которые вызываются в ответ на указанное событие.

Примеры и задания

Чтобы больше узнать о том, какими были самые первые компьютеры, посетите домашнюю страницу Дэйва Николса (Dave Nichols) по адресу <http://www.geocities.com/SiliconValley/Lakes/5705/360.html>. Кстати, о компьютерах. Я как-то видел тенниску, на которой было написано: "Мне удалось создать проблему двухтысячного года". Ну а для тех, кто делает бизнес на компьютерах и программах, проблема 2000-го года состояла в том, что она закончилась.

18-й час

Графические объекты библиотеки Tk, I

В этой главе мы познакомимся с окнами и графическими объектами библиотеки Tkinter. Прочитав главу до конца, Вы сможете отличить окно от графического объекта; различать типы графических объектов; использовать функции обработки событий с объектами Tkinter.

Окна и графические объекты

Отличить окно от графического объекта очень просто. Окно — это часть экрана, ограниченная каким-либо образом, обычно с помощью видимых границ. За выводение окна на экран отвечает либо диспетчер окон (в X Windows), либо сама операционная система (в Windows). Графический объект — это особый вид окна, снабженного функциональностью. (Не звучит ли это для Вас знакомым?) Обычно графические объекты меньше окон. Примеры окна и графического объекта приведены на рис. 18.1. В действительности различие между окном и графическим

объектом условно. Можно сказать, что графический объект — это особый вид окна, снабженного методами.

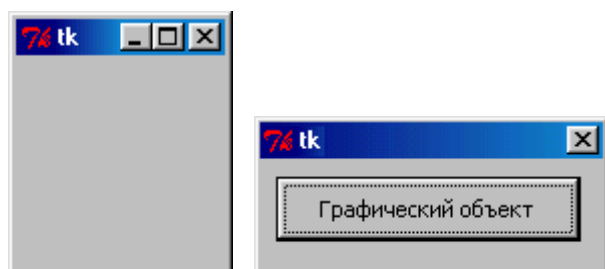


Рис. 18.1. Окно и графический объект

Слева на рис. 18.1 показано обычное окно, которое не обладает никакой функциональностью, кроме того, что занимает часть экрана. В этом окне пока что нет ни командных кнопок, ни строки меню. Чтобы отобразить это окно, достаточно ввести и выполнить следующий предельно простой код:

```
from Tkinter import *
root=Tk()
root.mainloop()
```

Единственный способ завершить эту программу — щелкнуть на маленькой кнопке с крестиком в верхнем правом углу окна. Объект справа, это тоже окно, но в него вставлен графический объект — кнопка с надписью "Графический объект". Правда, щелчок на кнопке не приводит ни к каким действиям. (Пример кнопки с назначенной функцией мы рассматривали в предыдущей главе). Код программы отображения данной кнопки показан в листинге 18.1.

Листинг 18.1. Программа tkt3.py

```
#!C:\PYTHON\PYTHON.EXE
from Tkinter import *
import sys
root=Tk()
button=Button(root)
button["text"]="Графический объект"
button.pack()
root.mainloop()
```

*Прим. В. Шипкова: позволю себе ещё раз напомнить, что если программист не желает, что бы пользователь увидел абракадабру вместо надписи, то вместо

```
button["text"]="Графический объект"
```

следует написать

```
button["text"]=u"Графический объект"  
или изначально использовать кодировку utf8
```

Как уже говорилось, графические объекты характеризуются состоянием и функциональностью. Они владеют методами, которые можно вызывать, и свойствами, которые можно устанавливать. Если быть точным, то корневое окно, показанное на рис. 18.1, слева, в действительности таковым не является. Это инкапсуляция окна в библиотеке Tkinter, первое производное от корневого окна. Все графические объекты библиотеки Tkinter являются дочерними объектами корневого окна, или, как ещё говорят, — окна верхнего уровня. Это окно обладает базовыми свойствами и функциональностью, которые наследуются дочерним объектам. В деталях наследование объектов в Python мы рассматривали в главах 10 и 11. Все графические объекты библиотеки Tkinter являются обычными программными объектами, характеризующимися состоянием (свойства) и функциональностью (методы). С помощью наследования и изменения функциональности можно создавать такие графические объекты, которые отвечают вашим требованиям.

Подробную информацию обо всех графических объектах библиотеки Tkinter с указанием всех их свойств и методов можно найти на странице <http://www.pythonware.com/library/tkinter/tclass/index.htm>.

В следующих разделах мы подробно рассмотрим все графические объекты библиотеки Tkinter с примерами простейших программ для их вывода. В главе 19 мы продолжим рассмотрение графических объектов и завершим главу созданием полноценного приложения с интерфейсом GUI. Простейшим из всех графических объектов, пожалуй, является кнопка, показанная на рис. 18.2. Ещё один пример кода вывода кнопки, изображенной на этом рисунке, показан в листинге 18.2.

Объект кнопка

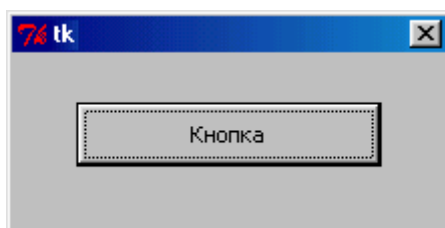


Рис. 18.2. Объект кнопка

```

from Tkinter import *
import sys

def die(event):
    sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>",die)
button.pack()

root.mainloop()

```

Код программы tkbutton.py напоминает рассмотренный выше код программы tkt3.py в листинге 18.1, если не считать добавленную функцию завершения программы. Но кнопку можно создать и другим способом. В Tkinter предусмотрена процедура создания графического объекта с синтаксисом вызова функции с параметрами. Вот как можно с помощью одной строки создать кнопку:

```
button = Button(root, text="Кнопка" ,command=die)
```

Этой строкой можно заменить строки 10-12 в листинге 18.2. После создания объекта все его свойства можно устанавливать, используя показанный выше синтаксис словаря: *ключ=значение*. Единственное, что следует при этом учитывать, так это то, что опция *command* ожидает имя такой функции обработки события, которая не имеет параметров. Таким образом, если в листинге 18.2 определение объекта кнопки мы заменим на альтернативный одностроочный вариант, то также придётся внести изменения в определение функции *die(event)* и удалить из неё параметр *event*. В данном случае такие изменения внести не сложно, правда, и сокращение длины кода будет незначительным. Тем не менее альтернативный метод определения графических объектов следует держать на вооружении. Иногда он оказывается весьма эффективным, особенно когда в объекте нужно установить много свойств. С другой стороны, назначение функций обработки событий с помощью метода *bind()* предпочтительнее. Использование выражения *command=имя_функции* ограничивает Ваши возможности назначать только функции без параметров и только единственному событию, тогда как с помощью метода *bind()* функции обработки можно назначить всем доступным событиям объекта, причём выполнением этих функций можно управлять с помощью параметра.

Графический объект холст (рис. 18.3) кажется скучным и бесполезным. В действительности это очень интересный объект, позволяющий создавать в нем произвольные рисунки. О том, как это делается, мы узнаем в главах этой части, посвященных графическим функциям библиотеки Tkinter.

Объект холст



Рис. 18.3. Объект холст

Простейшая программа, с помощью которой в одном окне можно вывести кнопку с холстом, показана в листинге 18.3.

Листинг 18.3. Программа `tkcanvas.py`

```
from Tkinter import *
import sys

def die(event):
    sys.exit(0)

root=Tk()
button=Button(root)
button["text"]="Button"
button.bind("<Button>",die)
button.pack()
canvas=Canvas(root)
canvas["height"]=64
canvas["width"]=64
canvas["borderwidth"]=2
canvas["relief"]=RAISED
canvas.pack()
root.mainloop()
```

Свойства `height`(высота), `width`(ширина) и другие можно устанавливать в параметрах, как было показано выше на примере объекта кнопки. Впрочем, и в этом случае выигрыш в уменьшении числа строк кода будет незначительным. Установка свойств в параметрах имеет явное преимущество при определении окон сообщений, которые мы рассмотрим в конце этой главы.

Объект флажок

Флажок, пример которого показан на рис. 18.4, используется в тех случаях, когда от пользователя нужно получить ответ да-нет. Связанные с ними переменные называют переключателями, поскольку для них возможны только два состояния: включена и выключена.

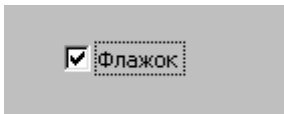


Рис. 18.4. Объект флажок

В листинге 18.4 показано, как можно создать флажок.

Листинг 18.4. Программа `tkcheckboxbutton.py`

```
from Tkinter import *
import sys

def die(event):
    sys.exit(0)

root=Tk()
button=Button(root)
button["text"]="Button"
button.bind("<Button>",die)
button.pack()
checkboxbutton=Checkbutton(root)
checkboxbutton["text"]="Checkbutton"
checkboxbutton.pack()

root.mainloop()
```

Следует учесть, что внешний вид большинства графических объектов, если не всех, зависит от операционной системы компьютера. Примеры, представленные в этой книге, были получены на компьютере под управлением Windows. В результате выполнения тех же программ на компьютерах Mac или с системой UNIX будут получены графические объекты, несколько отличающиеся по стилю от приведенных, хотя они вполне узнаваемы. Пример флажка в стиле UNIX показан на рис. 18.5.

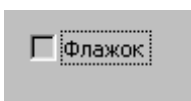


Рис. 18.5. Флажок в стиле UNIX

Объект текстовое поле (рис. 18.6) принимает текст, введённый пользователем. В поле можно ввести только одну строку, тогда как объект поле редактора, который мы рассмотрим в следующей главе, может принимать несколько строк. В листинге 18.5 показан простейший способ создания текстового поля.

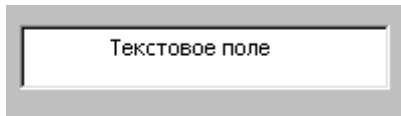


Рис. 18.6. Объект текстовое поле

Листинг 18.5. Программа tkenter.py

```
#!/usr/local/bin/python

from Tkinter import *
import sys

def die(event):
    print entry.get()
    sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>", die)
button.pack()

entry = Entry(root)
entry.insert(0, "Entry")
entry.pack()

root.mainloop()
```

***Прим. В. Шипкова:** обратите внимание на то, как описано положение исполняемого модуля Питона в стиле UNIX. без этой строки, программа, скорее всего, работать не будет. Для пользователей Windows указание местонахождения интерпретатора не актуально.

После того как щелчком на кнопке "Выход" Вы завершите выполнение программы, она выведет напоследок на печать содержимое текстового поля. Пример того, как получить текущее значение текстового поля и вывести его на печать, показан в строке 7 листинга 18.5. Для считывания значения используется метод `get()`. Методы `get()` и `set()` являются общими для всех графических объектов.

***Прим. В. Шипкова: здесь имелось в виду "вывод в окно консоли", а не "вывод на принтер".**

Объект рамка

Объект рамка (рис. 18.7) применяется для того, чтобы содержать в себе другие объекты. Его можно использовать также для выделения части экрана. Другими словами, рамки в окнах приложений используются либо как контейнеры, либо как средства оформления.

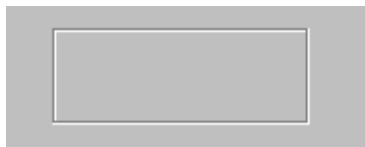


Рис. 18.7. Объект рамка

В листинге 18.7 показана простая программа, выводящая рамку. Обратите внимание, что свойства высоты и ширины рамки можно переустановить.

Листинг 18.6. Программа tkframe.py

```
from Tkinter import *
import sys

def die(event):
    sys.exit(0)

root=Tk()
button=Button(root)
button["text"]="Button"
button.bind("<Button>",die)
button.pack()
frame=Frame(root)
frame["height"]=64
frame["width"]=64
frame["background"]="white"
frame["borderwidth"]=2
frame["relief"]=RAISED
frame.pack()

root.mainloop()
```

Объект ярлык

Объект ярлык (рис. 18.8) является, пожалуй, одним из наиболее используемых.

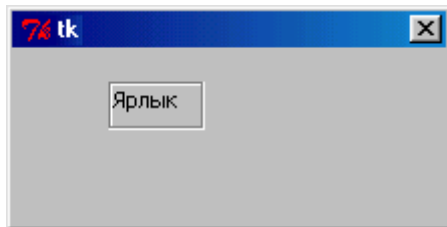


Рис. 18.8. Объект ярлык

Ярлык содержит неизменяемый текст. Создать его очень просто, как показано в листинге 18.7.

Листинг 18.7. Программа tklabel.py

```
from Tkinter import *
import sys

def die(event):
    sys.exit(0)

root=Tk()
button=Button(root)
button["text"]="Button"
button.bind("<Button>",die)
button.pack()
labelx=Label(root)
labelx["height"]=1
labelx.pack()
label=Label(root)
label["text"]="Label"
label["borderwidth"]=1
label["relief"]=SOLID

label.pack()

root.mainloop()
```

В этой программе для зрительного выделения ярлыка на экране используется дополнительный ярлык без текста `labelx`. То же самое можно было бы сделать с помощью объекта рамки.

Объект список

Объект список (рис. 18.9) используется для того, чтобы предоставить пользователю на выбор множество опций. При создании списков всегда проще использовать обычный способ определения, чем установку свойств с помощью параметров.



Рис. 18.9. Объект список (рисунок изменён В. Шипковым)

В листинге 18.8 показано, как создать список.

Листинг 18.8. Программа `tklistbox.py`

```
from Tkinter import *
import sys

elements=[u"Опция5", u"Опция4", u"Опция3", u"Опция2",
u"Опция1"]

def die(event):
    sys.exit(0)

root=Tk()
button=Button(root)
button["text"]=u"Выход"
button.bind("<Button>",die)
button.pack()
labelx=Label(root)
labelx["height"]=1
labelx.pack()

listbox=Listbox(root)
for i in elements :
    listbox.insert(0, i)
    listbox.pack()

root.mainloop()
```

*Прим. В. Шипкова: хоть названия элементов в списке приведены в обратном порядке, на самом деле Питон их чинтает с конца, поэтому список будет выглядеть к в прямом

перечислении, а не наоборот, как этого бы следовало ожидать.

Цикл `for` в строке 21 вводит текстовые значения всех опций списка. В действительности существует множество способов введения опций, это только один из них.

Переменные библиотеки Tkinter

Прежде чем использовать встроенные переменные графических объектов, следует уяснить, что все переменные библиотек Tk/TCL являются строковыми. Они используются во всех тех случаях, когда нужно организовать взаимодействие Python и Tkinter, т.е. в тех случаях, когда графическому объекту необходима переменная, которую он мог бы изменять, используется специальный набор встроенных переменных библиотеки Tkinter. В действительности эти переменные являются не просто строками, а объектами. Все они унаследованы от родительского класса `Variable`, который содержит ряд базовых методов и атрибутов, свойственных всем другим переменным графических объектов. Каждая такая переменная знает, как устанавливать, обрабатывать и возвращать данные только определённого типа. Список переменных библиотеки Tkinter, их описание и примеры создания показаны в табл. 18.1. (Не удивляйтесь, что они выглядят как функции, ведь они все-таки объекты, впрочем, как и все остальные переменные в Python.)

Таблица 18.1. Переменные библиотеки Tkinter

Переменная	Описание	Пример
<code>StringVar()</code>	Сохраняет строки	<code>x = StringVar("Текст")</code>
<code>IntVar()</code>	Сохраняет целые значения	<code>x = IntVar(42)</code>
<code>DoubleVar()</code>	Сохраняет значения с плавающей запятой	<code>x = DoubleVar(3.14159)</code>
<code>BooleanVar()</code>	Сохраняет значения <code>true</code> и <code>false</code> (истинно и ложно)	<code>x = BooleanVar("true")</code>

Чтобы вернуть значение любой из этих переменных, используется метод `get()`, например `print x.get()`. Для изменения значения используется метод `set()`, например `x.set(2*pi)`. Как Вы увидите ниже, обмен данными со всеми графическими объектами (как передача, так и возвращение) осуществляется только с помощью переменных библиотеки Tk.

Объекты меню и кнопка меню

Объект меню сам по себе не отображается на экране. Он обязательно должен содержать объекты кнопок меню. Объект меню можно представить себе в виде рамки, приспособленной исключительно для хранения объектов кнопок меню. Существуют два основных вида меню: строка меню и меню опций. Строка меню выглядит так, как показано на рис. 18.10.



Рис. 18.10. Строка меню

Вы видели подобную строку меню во всех приложениях для Windows. В листинге 18.9 показано, как её можно создать.

Листинг 18.9. Программа `tkmenu.py`

```
from Tkinter import *
import tkMessageBox
import sys

def die():
    sys.exit(0)

def callee():
    print u"Я был вызван!"

def about():
    tkMessageBox.showinfo("tkmenu", u"Это tkmenu.py \
        Версия 0")

root=Tk()
bar=Menu(root)

filem = Menu(bar)
filem.add_command(label=u"Открыть...", command=callee)
filem.add_command(label=u"Новый...", command=callee)
filem.add_command(label=u"Сохранить", command=callee)
filem.add_command(label=u"Сохранить как...", \
    command=callee)
filem.add_separator()
```

```

filem.add_command(label=u"Выход", command=die)

helpm = Menu(bar)
helpm.add_command(label=u"Индекс...", command=callee)
helpm.add_separator()
helpm.add_command(label=u"О программе", command=about)

bar.add_cascade(label=u"Файл", menu=filem)
bar.add_cascade(label=u"Помощь", menu=helpm)

root.config(menu=bar)
root.mainloop()

```

*Прим. В. Шипкова: прошу обратить внимание на строку:
print u"Я был вызван!"

Я указал Python, что строка является юникодом, хотя на самом деле эта строка может быть В ЛЮБОЙ КОДИРОВКЕ. Питон старательно переведёт её в юникод, а затем в классический ASCII (т. е. кодировку пригодную для консоли). Поэтому при выводе строки на консоль будет отображаться ПРАВИЛЬНЫЙ текст. Говоря проще, Питон – это сила! :)

В строках 13, 14 показан особый вид функции обработки события, которая возвращает сообщение после выбора команды "О программе". В этом случае вызывается графический объект окна сообщения, о котором мы поговорим далее в этой главе. В строке 19 создаётся объект меню filem, который является контейнером для ряда объектов кнопок меню, добавляемых в объект меню в строках 20-25. В строке 24 добавляется разделитель, который на экране выглядит как простая линия, отделяющая последнюю опцию "Выход" ото всех остальных опций меню "Файл". Как видите, нам не пришлось даже напрямую вызывать Menubutton(). Вместо этого вызова используется встроенный метод объекта меню add_command(). Вызов функции Menubutton() также возможен, но наш подход проще.

В строках 27-30 создаётся второй объект меню helpm, а в строках 32, 33 два объекта меню добавляются в объект строки меню bar, который был произведён от корневого окна в строке 17. Для добавления объектов меню в строку меню используется метод add_cascade(), который в действительности добавляет не сами объекты меню, а ярлыки, связанные с соответствующими объектами. Щелчок на ярлыке раскрывает связанное с ним меню, пример которого показан на рис. 18.11.

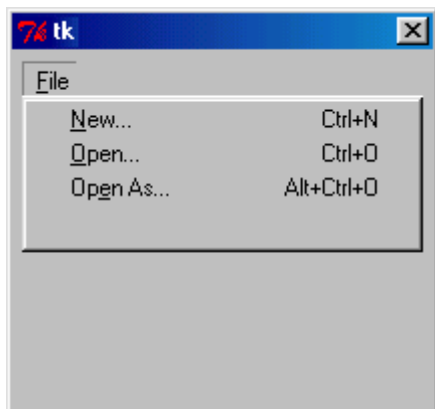


Рис. 18. 11. Раскрывающееся меню

Можно назначить опциям раскрывающегося меню другое вложенное раскрывающееся меню, которое будет содержать свои объекты `Menubutton` (кнопки меню). Но я лично не сторонник такого стиля. Меню чувствительно к перемещению мыши, а создание сложной системы вложенных меню не лучшим образом реализует эту зависимость. Не знаю, как Вас, а меня всегда раздражают ненужные мне подменю, всплывающие на экране, когда я провожу указателем по опциям главного меню. Зато когда мне нужно что-то выбрать в подменю, оно предательски исчезает, из-за того что указатель соскользнул на главное меню. Мне гораздо больше по душе стиль, когда команды меню открывают диалоговые окна с наборами дополнительных опций. Кстати, по общепринятому соглашению за такими командами меню ставится многоточие, как показано на рис. 18.11 на примере опции Создать... . Те опции, которые открывают вложенные подменю, принято выделять стрелками. Таким образом, объекты меню помимо кнопок могут содержать объекты вложенных меню, флажки, переключатели и разделители.

Меню опций (рис. 18.12) создаётся несколько сложнее. Хотя это и не обязательно, но обычно для установки ярлыка меню опций используется переменная графических объектов `Tkinter`. Если строка меню, как правило, располагается в верхней части окна программы или в виде плавающего диалогового окна, то меню опций может располагаться где угодно. В листинге 18.10 представлен код программы, создающей меню опций, показанное на рис. 18.12.

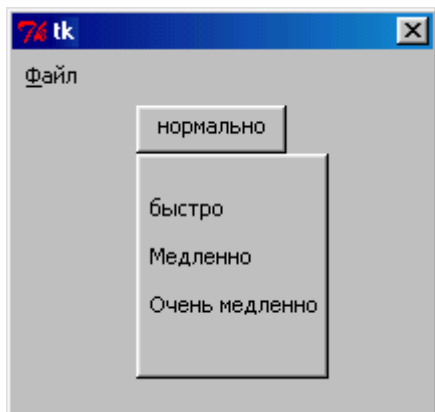


Рис. 18.12. Меню опций

Листинг 18.10. Программа tkoptionmenu.py

```
from Tkinter import *
import tkMessageBox
import sys

def die():
    global xx
    print xx.get()
    sys.exit(0)

def callee():
    print "I was called; few are chosen"

def about():
    tkMessageBox.showinfo("tkmenu", "This is tkmenu.py\
    Version 0")

root=Tk()
bar=Menu(root)

file=Menu(bar)
file.add_command(label="Open...", command=callee)
file.add_command(label="New...", command=callee)
file.add_command(label="Save", command=callee)
file.add_command(label="Save as...", command=callee)
file.add_separator()
file.add_command(label="Exit", command=die)

helpm=Menu(bar)
helpm.add_command(label="Index...", command=callee)
helpm.add_separator()
helpm.add_command(label="About", command=about)

bar.add_cascade(label="File", menu=file)
bar.add_cascade(label="Help", menu=helpm)
```

```
root.config(menu=bar)
frame = Frame(root)
frame.pack()
xx = StringVar(frame)
xx.set("slow")

fm=OptionMenu(frame, xx, "slow", "slower", "slowest", \
    "even slower")
fm.pack()
root.mainloop()
```

Как видите, данная программа выводит ту же строку меню, что и программа tkmenu.py из листинга 18.9. Но в программу tkoptionmenu.py добавлены дополнительные строки кода, создающие меню опций. В строке 40 создается строковая переменная графического объекта Tkinter, которая затем используется в строке 43. Вторым аргументом в вызове OptionMenu() должна быть переменная Tkinter, которой в строке 41 присваивается значение "нормально". Таким образом, устанавливается значение по умолчанию для меню опций. При выборе другой опции меню автоматически изменяется значение переменной xx. В строках 8, 9 показано, как возвратить это значение и вывести на печать.

Объекты окон сообщений

Окна сообщений являются *монопольными* диалоговыми окнами, которые всплывают поверх основного окна приложения и прерывают выполнение программы до тех пор, пока пользователь не щелкнет на одной из кнопок, представленных в окне. Кнопкой, выбираемой по умолчанию, для большинства окон сообщений является ОК. Но это не обязательно. Кроме того, есть окна сообщений, где эта кнопка вообще отсутствует, как, например, в окнах типа "Да-Нет", в которых представлены только две кнопки: Да и Нет. Свойства окон сообщений проще всего устанавливать как параметры функции tkMessageBox.showinfo(). Пример такого подхода Вы уже видели в строке 16 листинга 18.10. Иногда полезно использовать непосредственно функцию MessageBox(), например при создании окна вопроса, которое мы рассмотрим далее в этой главе. Обратите внимание, что все окна сообщений, показанные на рисунках ниже, были получены в системе Windows. Те же окна в UNIX будут выглядеть немного иначе. Они будут отличаться хотя бы тем, что в UNIX используются черно-белые пиктограммы, а не цветные, как в Windows. Но приведенные ниже функции подходят для любой операционной системы и платформы.

Информационное окно сообщения

Код создания информационного окна сообщения показан в листинге 18.11.

Листинг 18.11. Программа tkmessage.info.py

```
from Tkinter import *
import tkMessageBox
import sys

def die(event):
    tkMessageBox.showinfo("tkMessageBox", "tkMessageBox.showi
nfo")
    sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>",die)
button.pack()

root.mainloop()
```

Первый параметр функции `showinfo()` (см. строку 8) устанавливает заголовок окна, а со вторым параметром передаётся текст сообщения, которое будет выведено в окне. На рис. 18.13 показан результат выполнения программы `tkmessage.info.py`.

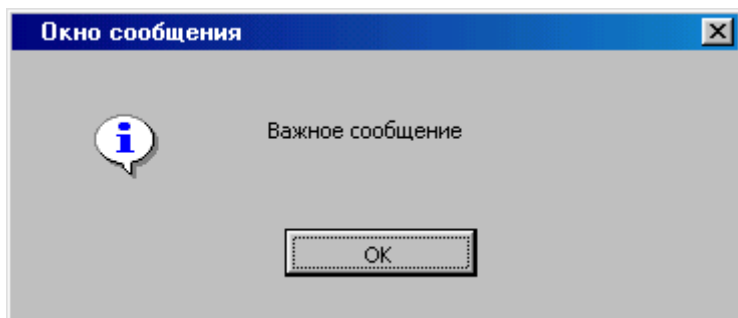


Рис. 18.13. Информационное окно сообщения

Большинство других окон сообщений такие же простые.

Предупреждающее окно сообщения

Предупреждающее окно сообщения показано на рис. 18.14.

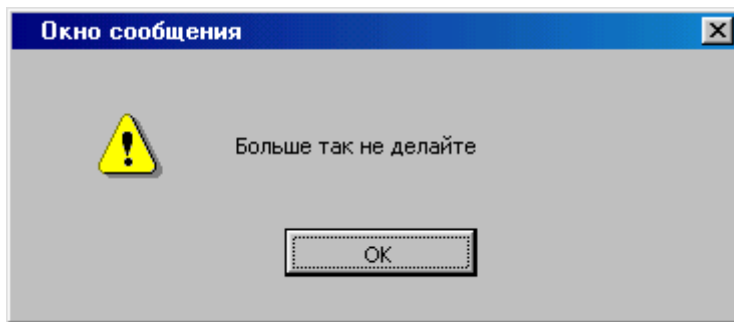


Рис. 18.14. Предупреждающее окно сообщения

В листинге 18.12 показан код программы предупреждающего окна сообщения.

```
from Tkinter import *
import tkMessageBox
import sys

def die(event):
    tkMessageBox.showwarning("tkMessageBox", "tkMessageBox.sho
warning")
    sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>", die)
button.pack()

root.mainloop()
```

Как видите, от предыдущего листинга эта программа отличается только строкой 8.

Окно сообщения об ошибке

Пример окна сообщения об ошибке показан на рис. 18.15.

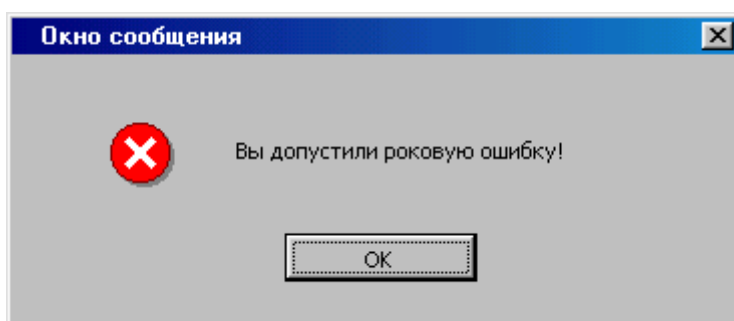


Рис. 18.15. Окно сообщения об ошибке

Окно сообщения об ошибке также содержит лишь одну кнопку ОК. Вы можете только прочитать сообщение и убрать его с экрана, иногда вместе с программой. Код вывода этого окна показан в листинге 18.13.

Листинг 18.13. Программа `tkmessage.error.py`

```
from Tkinter import *
import tkMessageBox
import sys

def die(event):
    tkMessageBox.showerror("tkMessageBox", "tkMessageBox.showe
rror")
    sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>", die)
button.pack()

root.mainloop()
```

И вновь, отличие коснулось только строки 8. Все три окна сообщений, рассмотренные выше, имели много общего. Все они содержали единственную кнопку ОК. От пользователя не ожидалось никаких действий, кроме того, что он должен прочесть сообщение и закрыть окно. В случае с сообщением об ошибке, вероятно, Вам следует установить автоматическое закрытие и само приложение, в котором произошла ошибка, если только Вы не уверены, что ничего страшного не произойдет и пользователь не потеряет свои данные. Окна сообщений следующей группы содержат больше одной кнопки, что позволяет пользователю сделать свой выбор.

Окно сообщения типа "Да-Нет"

Окно сообщения, показанное на рис. 18.16, предлагает пользователю прямо ответить на вопрос: да или нет.

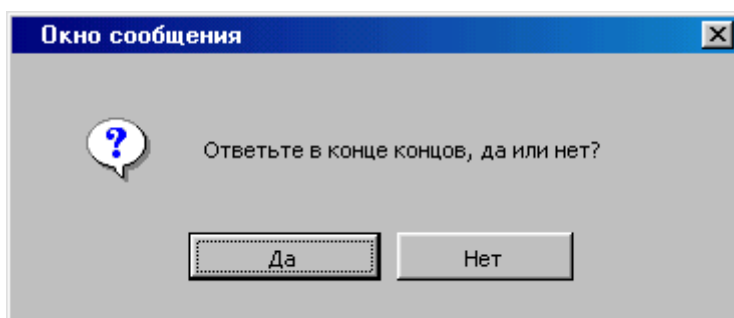


Рис. 18.16. Окно сообщения типа "Да-Нет"

Код программы показан в листинге 18.14.

Листинг 18.14. Программа `tkmessage.askyesno.py`

```
from Tkinter import *
import tkMessageBox
import sys

def die(event):
    response = tkMessageBox.askyesno("tkMessageBox",
    "tkMessageBox.askyesno")
    print response
    sys.exit(0)

root=Tk()
button=Button(root)
button["text"]="Button"
button.bind("<Button>",die)
button.pack()

root.mainloop()
```

После выполнения программы Вы увидите, что выбор кнопки "Да" возвратит значение 1, а выбор кнопки "Нет" – значение 0. Эти значения, соответствующие выбранной пользователем кнопке, возвращаются функцией `askyesno()` в строке 8. Далее Вы сможете установить путь выполнения программы в зависимости от выбора пользователя, воспользовавшись инструкцией `if`.

Окно сообщения типа "Ок-Отмена"

Следующее окно сообщения, показанное на рис. 18.17, также содержит только две кнопки, но называются они иначе.

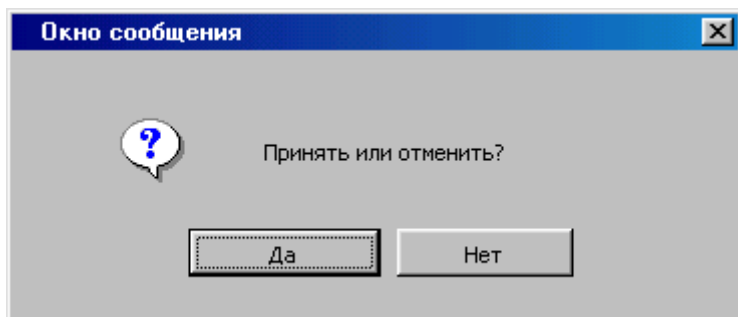


Рис. 18.17. Окно сообщения типа "Ок-Отмена"

В листинге 18.15 показан код программы вывода этого окна (как видите, данный код аналогичен для всех двухкнопочных окон сообщений).

Листинг 18.15. Программа `tkmessage.askokcancel.py`

```
from Tkinter import *
import tkMessageBox
import sys

def die(event):
    r = tkMessageBox.askokcancel("tkMessageBox",
    "tkMessageBox.askokcancel")
    print r
    sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>", die)
button.pack()

root.mainloop()
```

Окно сообщения типа "Повтор-Отмена"

Ещё одно двухкнопочное окно сообщения показано на рис. 18.18. Принцип его работы аналогичен предыдущему.

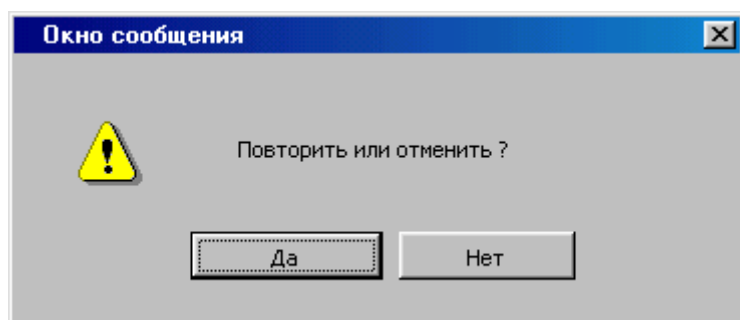


Рис. 18.18. Окно сообщения типа Повтор-Отмена

Код программы показан в листинге 18.16.

```
from Tkinter import *
import tkMessageBox
import sys

def die(event):
    r = tkMessageBox.askretrycancel("tkMessageBox", \
```

```

"tkMessageBox.askretrycancel")
print r
sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>", die)
button.pack()

root.mainloop()

```

Все три двухкнопочных окна сообщения используются совершенно одинаково. На вопрос, поставленный в сообщении, пользователь может ответить щелчком на одной из двух кнопок. Других возможностей закрыть окно у него просто нет, разве что выключить компьютер. В результате объект окна возвращает одно из двух значений: 0 или 1. Вам осталось только запомнить, что значение 0 соответствует выбору кнопок "Нет", "Отмена", "Пропустить", а значение 1 – кнопкам "ОК", "Да", "Повтор".

Окно вопроса

Чтобы отобразить трёхкнопочное окно, показанное на рис. 18.19, необходимо напрямую обратиться к методу `Message`, поскольку не существует специальных функций вывода окон, содержащих больше двух кнопок. Обратите внимание, что в этом случае весьма удобно использовать стиль назначения свойств в параметрах вызова функции.

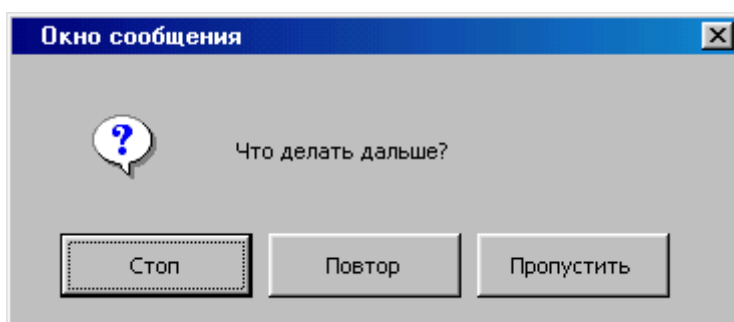


Рис. 18. 19. Окно вопроса Код программы показан в листинге 18.17.

Листинг 18.17. Программа `tkmessage.askquestion.py`

```

from Tkinter import *
import tkMessageBox
import sys

```

```

def die(event):
    x = tkMessageBox.Message(root,\
        type=tkMessageBox.ABORTRETRYIGNORE, \
        icon=tkMessageBox.QUESTION,\
        title="tkMessageBox",\
        message="tkMessageBox.askquestion")
    r = x.show()
    tkMessageBox.showinfo("Reply", r )
    sys.exit(0)

root = Tk()
button = Button(root)
button["text"] = "Button"
button.bind("<Button>",die)
button.pack()

root.mainloop()

```

Свойства окна сообщения устанавливаются в строках 8-11. Свойству `type` присвоено значение константы `ABORTRETRYIGNORE`, что означает трехкнопочное окно с кнопками "Стоп", "Повтор", "Пропустить". Ещё одна константа `QUESTION` устанавливает для свойства `icon` вид отображаемой пиктограммы – знак вопроса. Эти две константы определены в модуле `tkMessageBox`. Далее устанавливаются заголовок окна и текст сообщения соответственно для свойств `"title"` и `"message"`. Эти свойства ожидают получить строковые значения. В данном случае нам ещё необходимо выполнить две операции, которые раньше выполнялись автоматически функциями показа окон. Во-первых, нужно самостоятельно создать объект окна сообщения `x` (`x = tkMessageBox.Message()`), а во-вторых, вызвать метод `x.show()`. В строке 12 мы не только вызываем метод `show()`, но и присваиваем возвращенное им значение переменной `r`. Затем мы используем специальную функцию показа окна сообщений `showinfo()`, чтобы отобразить значение выбранной кнопки на экране. Выполните программу и обратите внимание на то, что `r` является строкой, а не числом. Это ещё одно отличие между использованием метода `Message` и специальной функции показа окна сообщения.

В действительности такой подход можно использовать для показа всех окон сообщения. Специальные функции выполняют лишь роль удобного интерфейса между пользователем и методом `Message()`.

Резюме

Мы рассмотрели различные типы базовых графических объектов библиотеки `Tkinter` – от кнопки до различных видов

окон сообщений. В следующей главе мы завершим эту тему, рассмотрев такие графические объекты, как переключатель и поле редактора, после чего попытаемся создать полнофункциональное приложение, демонстрирующее использование всех рассмотренных объектов в графическом пользовательском интерфейсе.

Практикум

Вопросы и ответы

Почему одни и те же графические объекты по-разному¹ выглядят в Windows, X Windows и Mac?

Первоначально графические объекты в системах Windows и X Windows выглядели почти одинаково, поскольку корнями уходили к общему предку — объектам, разработанным в компании HP. Но в дальнейшем они развивались разными путями. В X Windows они выглядят более упрощенно в так называемом стиле *Motif*. На графические объекты Windows сильное влияние оказали разработки компании Macintosh, где использовались графические объекты, разработанные в исследовательской лаборатории Palo Alto компании Xerox (Xerox's Palo Alto Research lab). В сочетании с классами Windows эти объекты видоизменялись от версии Windows 3.1 до Windows 95. Изначально в библиотеке Tk/TCL использовались графические объекты, которые выглядели совершенно одинаково во всех системах и на всех платформах. Но затем были разработаны объекты, отвечающие стилю текущей системы. Благодаря этому поддерживается эмоциональная сторона свойства переносимости программных продуктов на языке Python, т.е. программы не только успешно выполняются независимо от операционной системы, но и пользовательский интерфейс выглядит как родной.

В приложениях для Windows часто применяются списки, снабжённые полосой прокрутки. Что это такое — графический объект или их комбинация, и можно ли нечто подобное сделать в Python?

Такой объект называется комбинированным списком. В Tkinter нет аналогичного графического объекта, который можно было бы непосредственно использовать, но в принципе его можно разработать самостоятельно. Наиболее удачной в этом плане мне кажется разработка Грега Мак-Фарлана (Greg McFarlane) из Австралии, который собрал целую коллекцию комбинированных графических объектов и назвал её Python Megawidgets. В этой коллекции можно найти и комбинированный список. Посетите домашнюю страницу Python Megawidgets по адресу <http://www.dscpl.com.au/pmw/>. Сам Грег так

охарактеризовал свою коллекцию: "Она построена на базовых графических объектах Tkinter и содержит классы и библиотеки легко настраиваемых и расширяемых графических объектов. В коллекции Вы можете найти различные варианты записных книжек, раскрывающихся и комбинированных списков, полос прокрутки, диалоговых окон и многое другое". Следует добавить, что классы всех этих объектов написаны исключительно на Python без блоков на языке C, что чрезвычайно важно для тех пользователей, у которых нет компилятора C или C++.

Контрольные вопросы

1. Какие графические объекты работают по принципу переключателя?
 - а) Кнопка.
 - б) Список.
 - в) Флажок.
 - г) Текстовое поле.
2. Какая разница между строкой меню и меню опций?
 - а) Строка меню расположена в верхней части главного окна приложения, тогда как меню опций находится внизу.
 - б) Строка меню раскрывается вниз, а меню опций – в сторону.
 - в) Меню опций может быть расположено в любой части окна приложения, тогда как строка меню всегда находится вверху окна под строкой заголовка.
 - г) Это разные названия одного и того же объекта.

Ответы

1. в. Флажок работает как переключатель, так как имеет только два состояния – выбран и не выбран.
2. в. Меню опций может быть расположено в любой части окна приложения, тогда как строка меню всегда находится вверху окна под строкой заголовка.

Примеры и задания

Разыщите информацию об истории развития графического пользовательского интерфейса. Начните с Web-страницы Майкла Хофмана (Michael S. Hoffman) The UNIX GUI Manifesto (Манифест UNIX GUI), расположенной по адресу <http://www.cybtrans.com/infostrc/unixgui.htm>.

В предыдущей главе мы начали изучение графических объектов библиотеки Tkinter. Сейчас мы продолжим рассмотрение этой темы. Прочитав данную главу до конца, Вы сможете узнать дополнительные графические объекты, с которыми ещё не знакомы; комбинировать различные объекты в графическом интерфейсе небольшого приложения; создавать сложные полнофункциональные приложения, такие как tkeditor.

Объект переключатель

Вы, наверное, ещё помните переключатели каналов и режимов на радиоаппаратуре, которые имели вид утапливаемых кнопок. Сейчас их почти повсеместно заменили сенсорные, но лет 20 назад практически вся радиоаппаратура пестрела такими кнопками (чем больше – тем круче). Именно из-за сходства с этими кнопками для данного типа графических объектов в английской терминологии было выбрано название *Radio Button*. В чем же это сходство? Вспомните, на радиоприемнике Вы нажимаете кнопку одного канала, и тут же другая кнопка, которая была нажатой до того, стремительным щелчком возвращается в исходную позицию в ряду с другими кнопками, т.е. сколько бы ни было кнопок в одном ряду, нажатой может быть только одна, ведь нельзя же слушать одновременно два канала. Кнопки переключателя (рис. 19.1) ведут себя точно так же. Вот почему в набор переключателей всегда входит несколько кнопок. Одна кнопка переключателя мертва, как муравей без муравейника. Как и в случае с другими графическими объектами, переключатели в системе UNIX ведут себя точно так же, только выглядят немного иначе (рис. 19.2).

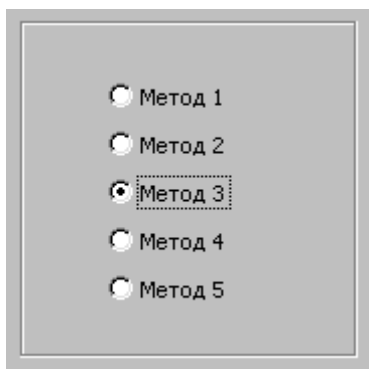


Рис. 19.1. Объект переключатель



Рис. 19.2. Объект переключатель в стиле UNIX

Но даже в Windows можно настроить вид кнопок переключателя, чтобы они выглядели как обычные кнопки (рис. 19.3).

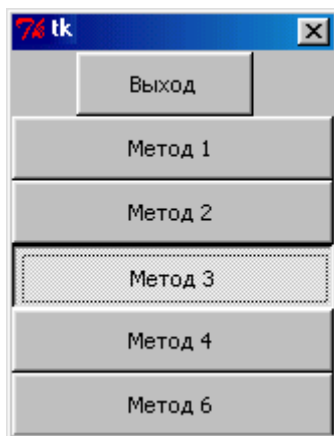


Рис. 19.3. Альтернативный стиль отображения кнопок переключателя

Проанализировав код программы вывода переключателя, который показан в листинге 19.1, можно понять, что при запуске программы без аргументов устанавливается стиль кнопок переключателя, заданный по умолчанию (круглые кнопки). При вводе любого аргумента устанавливается альтернативный стиль, показанный на рис. 19.3.

Листинг 19.1. Программа `tkradiobutton.py`

```
from Tkinter import *
import sys

def die(event):
    global v, periods
    print periods[v.get()][0]
    sys.exit(0)

root = Tk()
```

```

button = Button(root)
button["text"] = "Quit"
button.bind("<Button>",die)
button.pack()

v = IntVar()
v.set(2)

periods = [
    ("kin", 0),
    ("uinal", 1),
    ("tun", 2),
    ("katun", 3),
    ("baktun", 4),
]

if len(sys.argv) > 1:
    indicator = 0
    filler=X
    expander=1
else:
    indicator = 1
    filler=None
    expander=0
for t, m in periods :
    b = Radiobutton(root,text=t, variable=v, value=m,\
        indicatoron=indicator)
    if indicator == 1 :
        b.pack(anchor=W)
    else :
        b.pack(expand=expander,fill=filler)

root.mainloop()

```

***Прим. В. Шипкова:** обратите внимание, как красиво построен цикл **for** в конце приведённого листинга - в качестве итераторов используются две(!!!) переменных - **t** и **m**. Вряд ли Вы где увидите нечто подобное.

В строках 17, 18 используется переменная библиотеки Tk, в этот раз целочисленная. В данном случае использование числовой переменной более логично, хотя мы могли воспользоваться и строковой переменной, изменив наш набор **methods**, например, следующим образом: ("Метод 1", "mth1"), ("Метод 2", "mth2") и т.д. Чтобы вывести затем выбранное значение на экран, можно было бы воспользоваться методом **v.get()** или окном сообщения **showinfo()**.

Объект шкала

Этот объект шкала можно вывести на экран в горизонтальном (рис. 19.4) или вертикальном (рис. 19.5) положении.



Рис. 19.4. Горизонтальная шкала



Рис. 19.5. Вертикальная шкала

Объект шкала используется каждый раз, когда необходимо дать возможность пользователю выбрать значение в заданном диапазоне. В нашем примере на горизонтальной шкале можно выбрать значение в диапазоне от 0 до 100, т.е. 101 дискретное значение. В листинге 19.2 показан код вывода шкалы как в горизонтальном, так и в вертикальном положении.

Листинг 19.2. Программа tkyscale.py

```
from Tkinter import *
import sys
import string

def die(event):
    sys.exit(0)

def reader(s):
    f=string.atoi(s)
    f=f-32
    c=f/9.
    c=c*5.
    print "%s degrees F = %f degrees C" % (s, c)

root = Tk()
button = Button(root)
button["text"] = "Quit"
button.bind("<Button>",die)
button.pack()

scale1 = Scale(root, orient=HORIZONTAL)
scale1.pack()
scale2 = Scale(root, orient=VERTICAL,from_=-40, to=212,\
    command=reader)
```

```
scale2.pack()
```

```
root.mainloop()
```

Запустив программу на выполнение, Вы увидите, что вертикальная шкала настроена таким образом, чтобы преобразовывать значения градусов по Фаренгейту (в пределах от -40 F до 212 F) в градусы по Цельсию (соответственно от -40°C до 100°C). Нижняя граница -40° была выбрана потому, что это значение совпадает в обеих шкалах.

Исходные и расчётные значения динамически передаются программой в объект `stdout`, который выводит их на экран в окне терминала. Вероятно, это не самый элегантный подход к выводу данных в программе, использующей интерфейс GUI. Давайте внесём некоторые изменения в программу `tkscale.py`, чтобы ввод и вывод данных происходил в одном окне. Мы добавим несколько ярлыков, удалим ненужную горизонтальную шкалу и подпишем нашу программу в строке заголовка, чтобы сразу стало ясно, зачем она нужна. Изменённый код программы показан в листинге 19.3.

Листинг 19.3. Программа `tkscale2.py`

```
from Tkinter import *
import sys
import string

def die(event):
    sys.exit(0)

def reader(s):
    global label
    f=string.atoi(s)
    f=f-32
    c=f/9.
    c=c*5.
    flabel["text"]="Degrees Fahrenheit: %s" % (s)
    label["text"]="Degrees Celsius: %d" % (int(c))

root=Tk()
button=Button(root,width=25)
button["text"]="Quit"
button.bind("<Button>",die)
button.pack()

flabel=Label(root,text="Degrees Fahrenheit:")
flabel.pack(anchor=W)
```

```

scale2=Scale(root, orient=VERTICAL, from_=-
40, to=212, command=reader)
scale2.pack()
label=Label(root, text="Degrees Celsius:")
label.pack(anchor=W)

root.title("Fahrenheit to Celsius")

root.mainloop()

```

На рис. 19.6 показано, как теперь будет выглядеть окно приложения.

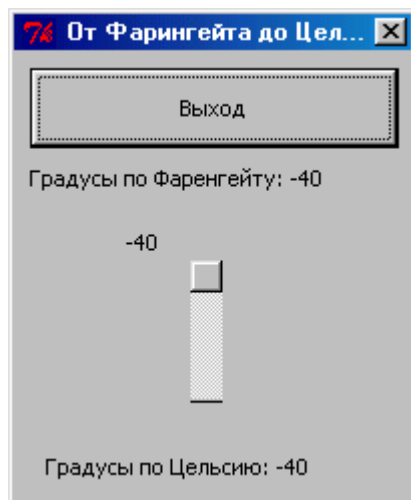


Рис. 19.6. Окно программы *tkscale2.py*

Поле редактора с полосой прокрутки

Объект поле редактора (рис. 19.7) предназначен для ввода пользователем многострочного текста. Этот объект снабжен функциями, поддерживающими редактирование и выделение текста с помощью мыши и комбинаций быстрых клавиш. Так, блок текста, выделенный указателем мыши, можно скопировать, вырезать и вставить, используя комбинации клавиш, характерные для данной системы. Например, в Windows для этого используются комбинации клавиш <Ctrl+C>, <Ctrl+X> и <Ctrl+V> соответственно. В листинге 19.4 показан минимальный код вывода объекта поле редактора.

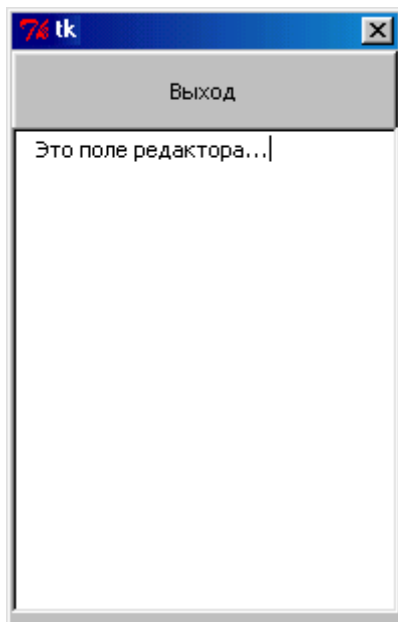


Рис. 19.7. Объект поле редактора

Листинг 19.4. Программа tktext.py

```
from Tkinter import *
import sys

def die(event):
    sys.exit(0)

root=Tk()
f=Frame(root)
f.pack(expand=1, fill=BOTH)
button=Button(f,width=25)
button["text"] = "Button"
button.bind("<Button>",die)
button.pack()
t = Text(f, width=25, height=10, relief=RAISED, bd=2)
t.pack(side=LEFT, fill=BOTH, expand=1)

root.mainloop()
```

В строке 16 свойству `bd` назначается ширина границы объекта текст. Имя `bd` — это сокращение от имени свойства `borderwidth`. Для некоторых свойств позволено использование как полных имён, так и стандартных сокращений, являющихся синонимами.

Полосы прокрутки

Полосы прокрутки используются для того, чтобы прокручивать вверх и вниз содержимое других объектов, например текст в поле редактора. Полоса прокрутки может

быть горизонтальной, но для неё в Tkinter назначено очень мало функций. Гораздо шире используется вертикальная полоса прокрутки, которая применяется в таких объектах, как `ScrolledText` (прокручиваемое поле редактора), `FileDialog` (диалоговое окно Файл) и др.

Пример вертикальной полосы прокрутки показан на рис. 19.8.

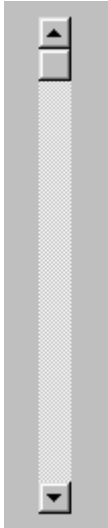


Рис. 19.8. Полоса прокрутки

Вместо того чтобы показывать код вывода отдельной полосы прокрутки, которая сама по себе нигде не используется, в листинге 19.5 показан код прокручиваемого поля редактора. Это комбинированный графический объект, т.е. объект, состоящий из нескольких базовых объектов. Ниже представлен простейший вариант комбинированного объекта.

Листинг 19.5. Программа `tkscrollbar.py`

```
from Tkinter import *
from ScrolledText import *
import sys

def die(event):
    sys.exit(0)

root=Tk()
f=Frame(root)
f.pack(expand=1, fill=BOTH)
button=Button(f,width=25)
button["text"]="Quit"
button.bind("<Button>",die)
button.pack()

st=ScrolledText(f,background="white")
st.pack()
```



```
root.mainloop()
```

Эта небольшая программа выводит на экран поле редактора. Безусловно, возможности использования данного редактора весьма ограничены. Вы даже не можете сохранить введенный текст. Тем не менее для решения подобной задачи в языке C потребовалась бы программа, состоящая из сотни строк. (В действительности так оно и есть, поскольку каждая строка кода Python-Tkinter вызывает выполнение целого каскада строк на языке C, положенных в основу используемых нами графических объектов.) Python – это язык программирования высокого уровня. Он освобождает Вас от решения таких рутинных низкоуровневых задач, как отслеживание текущего положения курсора и выбор способа вывода отдельных букв при нажатии клавиш. Комплексное решение множества низкоуровневых задач одной командой языка высокого уровня называется *абстракцией*. Пример поля редактора, выведенного на экран программой `tkscrollbar.py`, показан на рис. 19.9.

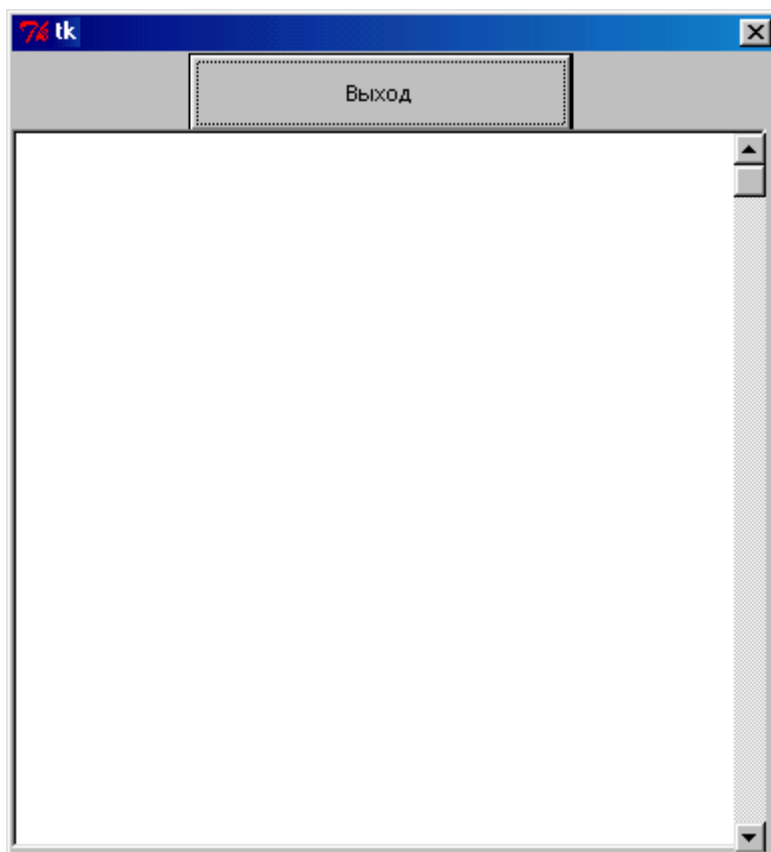


Рис. 19.9. Поле редактора с полосой прокрутки

Хватит плескаться на мелководье. Давайте создадим такой текстовый редактор, который позволит нам сохранять введенный текст. Безусловно, нам также следует добавить строку меню, в которой, помимо опции сохранения файла будет команда открытия информационного окна с данными о версии программы. В общем, мы все это уже умеем делать.

Единственное, чего Вы ещё не знаете, — где взять диалоговое окно для сохранения и открытия файлов. Полный код нашего текстового редактора показан в листинге 19.6.

*Прим. В. Шипкова: программа блокнота рабочая, но всё-таки сохранить русский текст Вы не сможете. Это связано с тем, что Питон сам плохо справляется с преобразованием юникода с ASCII, с кодами большими чем 128. Но это всё поправимо. ;)

Листинг 19.6. Текстовый редактор `tkeditor.py`

```
from Tkinter import *
from ScrolledText import *
import tkMessageBox
from tkFileDialog import *
import fileinput

st=None

def die():
    sys.exit(0)

def openfile():
    global st
    pl=END
    oname=askopenfilename(filetypes=[("Python files",\
        "*.py")])
    if oname:
        for line in fileinput.input(oname):
            st.insert(pl,line)

def savefile():
    sname=asksaveasfilename()
    if sname:
        ofp=open(sname,"w")
        ofp.write(st.get(1.0,END))
        ofp.flush()
        ofp.close()

def about():
    tkMessageBox.showinfo("Tkeditor", \
        "Simple tkeditor Version 0\n"\
        "Written 1999\n"\
        "For Teach Yourself Python in 24 Hours")

if __name__=="__main__":
```

```

global st
root=Tk()
bar=Menu(root)

filem=Menu(bar)
filem.add_command(label="Open...", command=openfile)
filem.add_command(label="Save as...", \
    command=savefile)
filem.add_separator()
filem.add_command(label="Exit", command=die)

helpm=Menu(bar)
helpm.add_command(label="About", command=about)

bar.add_cascade(label="File", menu=filem)
bar.add_cascade(label="Help", menu=helpm)
root.config(menu=bar)

f=Frame(root,width=512)
f.pack(expand=1, fill=BOTH)

st=ScrolledText(f,background="white")
st.pack(side=LEFT, fill=BOTH, expand=1)
root.mainloop()

```

На рис. 19.10 показан результат выполнения программы `tkeditor.py`. В поле редактора введён собственный код программы, и мы сохраняем программу в папке `C:\Python`. Диалоговое окно "Сохранение" вызывается с помощью функции обработки события `savefile()`.

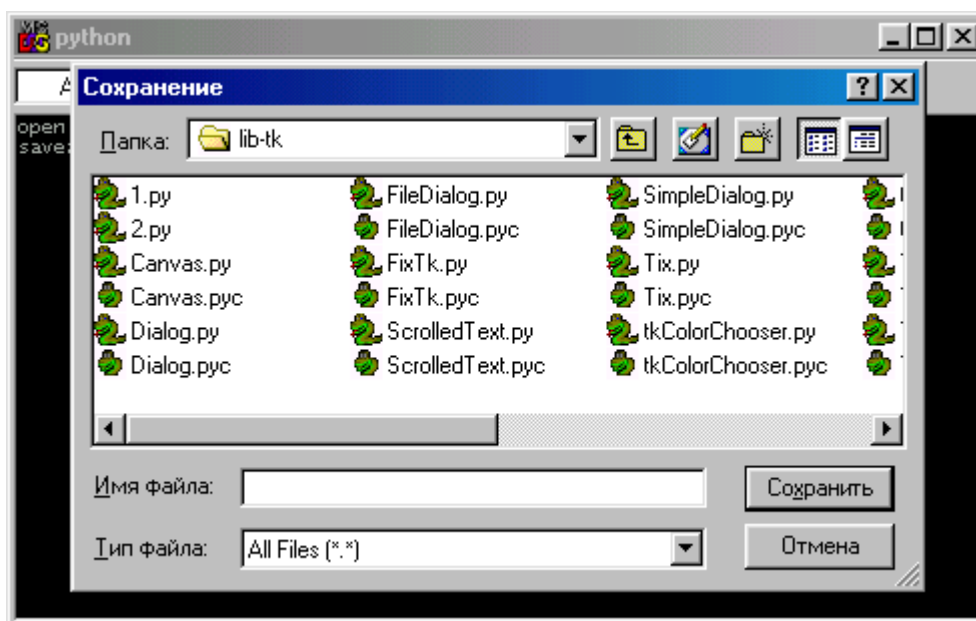


Рис. 19.10. Сохранение файла в редакторе `tkeditor.py`

В листинге 19.5 содержится 59 строк, включая пустые. Ещё пару лет тому назад программа подобного текстового редактора состояла бы из сотен строк, а десяток лет тому назад — из тысяч строк.

Конечно, для того чтобы этот редактор стал полноценным приложением, в него нужно ещё добавить много полезных средств и инструментов. Многие из них Вы сможете добавить сами. Например, хорошо было бы иметь возможность внедрять в текст графические изображения или сохранять файлы разных форматов. И сделать это действительно просто. Нужно только знать, где хранятся требуемые методы и объекты. А в этом как раз нет никакого секрета. На домашней странице Python Вы найдёте всю исчерпывающую информацию, а также ссылки на другие Web-страницы.

Окна верхнего уровня

К окнам верхнего уровня относятся главные окна приложений, которые содержат все остальные графические объекты. Так, в меню нашей программы текстового редактора `tkeditor.py` можно добавить команду Создать..., которая будет открывать новое пустое окно редактора, в точности соответствующее исходному окну (чуть позже мы действительно добавим эту опцию). Такая возможность существует в редакторе IDLE. Выберите в меню File опцию New window, и тут же на экране появится новое окно интерпретатора Python. В листинге 19.7 показан один из простейших способов открытия окон верхнего уровня.

Листинг 19.7. Программа `tktoplevel.py`

```
from Tkinter import *
import sys

wl=[]

def die():
    print "You created %d new toplevel widgets"%(len(wl))
    sys.exit(0)

def newwin():
    global root
    wl.append (Toplevel(root))

root=Tk()
bar=Menu(root)
filem=Menu(bar)
filem.add_command(label="New...", command=newwin)
```

```
filem.add_separator()
filem.add_command(label="Exit", command=die)

bar.add_cascade(label="File", menu=filem)
root.config(menu=bar)
root.mainloop()
```

Теперь Опустим программу на выполнение и выберем команду Создать... из меню Файл 4 раза. Результат показан на рис. 19.11.

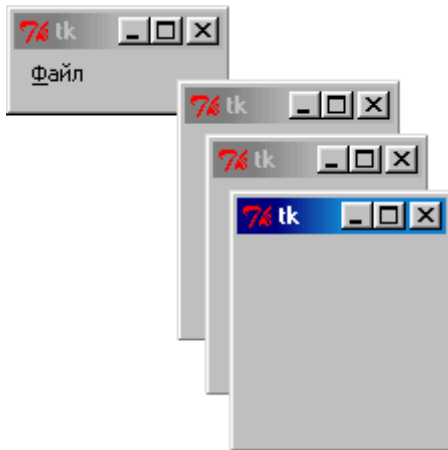


Рис. 19.11. Несколько окон верхнего уровня, открытые из окна программы *tktoplevel.py*

Программа работает, правда, нам мало проку от множества окон, которые ничего не умеют делать, кроме как занимать пространство на экране. Гораздо полезнее было бы иметь несколько дополнительных полнофункциональных окон. Например, если в нашу программу текстового редактора добавить возможность открывать новые окна, мы могли бы одновременно работать с несколькими текстовыми файлами. Эту задачу можно выполнить несколькими способами, но лучшими будет создание класса окна редактора, который можно будет реализовать в программе неограниченное число раз. Мало того, что таким путём мы сократим основной код программы до нескольких строк, но мы также можем сохранить нашу программу как модуль и импортировать её затем во все другие приложения, где возникнет необходимость в текстовом редакторе. Измененный код программы текстового редактора, в которой теперь появился класс окна редактора `editor` (практически весь код программы – строки 20-68), показан в листинге 19.8. Этот класс содержит ряд методов. Кроме того, в программе определены ещё три функции, которые не было смысла делать членами класса. Это функции `die()` (строка 12), `about()` (строка 15) и `neweditorf()` (строка 70).

```

from Tkinter import *
from ScrolledText import *
import tkMessageBox
from tkFileDialog import *
import fileinput

tl=[]
root=None

def die():
    sys.exit(0)

def about():
    tkMessageBox.showinfo("Tkeditor", \
        "Simple tkeditor Version 1\n"\
        "Written 1999\n"\
        "For Teach Yourself Python in 24 Hours")

class editor:
    def __init__(self, rt):
        if rt==None:
            self.t=Tk()
        else:
            self.t=Toplevel(rt)
        self.t.title("Tkeditor %d" % len(tl))
        self.bar = Menu(rt)

        self.filem = Menu(self.bar)
        self.filem.add_command(label="Open...", \
            command=self.openfile)
        self.filem.add_command(label="New...", \
            command=neweditor)
        self.filem.add_command(label="Save as...", \
            command=self.savefile)
        self.filem.add_command(label="Close", \
            command=self.close)
        self.filem.add_separator()
        self.filem.add_command(label="Exit", command=die)

        self.helpm = Menu(self.bar)
        self.helpm.add_command(label="About", command=about)

        self.bar.add_cascade(label="File", menu=self.filem)
        self.bar.add_cascade(label="Help", menu=self.helpm)
        self.t.config(menu=self.bar)

        self.f = Frame(self.t,width=512)
        self.f.pack(expand=1, fill=BOTH)

```

```

self.st = ScrolledText(self.f,background="white")
self.st.pack(side=LEFT, fill=BOTH, expand=1)

def close(self):
    self.t.destroy()

def openfile(self):
    pl=END
    oname = askopenfilename(filetypes=[("Python files",\
        "*.py")])
    if oname:
        for line in fileinput.input(oname):
            self.st.insert(pl,line)
            self.t.title(oname)

def savefile(self):
    sname=asksaveasfilename()
    if sname:
        ofp=open(sname,"w")
        ofp.write(self.st.get(1.0,END))
        ofp.flush()
        ofp.close()
        self.t.title(sname)

def neweditor():
    global root
    tl.append(editor(root))

if __name__=="__main__":
    root=None
    tl.append(editor(root))
    root=tl[0].t
    root.mainloop()

```

Обратите внимание, что когда мы открываем новый файл или сохраняем файл под другим именем, изменяется заголовок текущего окна. Эти изменения реализуются строками программы 59 и 68 с помощью метода окна верхнего уровня `title()`. Окна верхнего уровня содержат ещё много полезных методов, о которых Вы можете узнать в документации библиотеки Tkinter.

И опять-таки, обратите внимание, что вся программа текстового редактора с возможностью открытия нескольких окон реализована в 78 строках кода. Однажды в 1984 г. я потратил неделю на написание программы текстового редактора. Код программы составил около 10000 строк. Мне приходилось неоднократно выводить на печать весь код, потому что невозможно было удержать в голове, что делает тот или иной блок. Используемый мной язык программирования

не поддерживал никаких абстракций, поэтому приходилось учитывать такие технические мелочи работы компьютера, о которых сейчас страшно и подумать. Текстовый редактор, который мы сейчас получили и который я до этого спроектировал за считанные часы, несравнимо, мощнее и эффективнее моего старого проекта. На рис. 19.12 показано, как работает наш текстовый редактор.

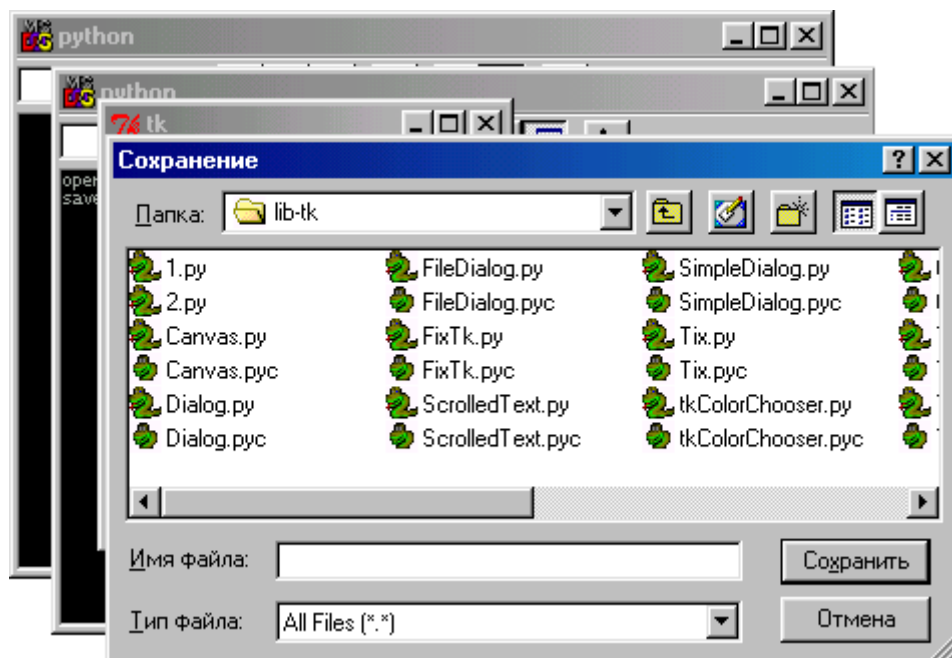


Рис. 19.12. Текстовый редактор с несколькими окнами редактирования

На этом мы завершим знакомство с различными графическими редакторами библиотеки Tkinter.

Резюме

Мы познакомились со всеми графическими объектами библиотеки Tkinter и написали две программы: `tkscale2.py` и `multi-editor.py`. Последняя программа – это полнофункциональный текстовый редактор, позволяющий открывать несколько окон для одновременного редактирования разных файлов. Что примечательно, весь код этой сложной программы уместился в 78 строках. В следующих главах мы изучим графические функции библиотеки Tkinter, позволяющие рисовать в объекте холст. Изучив все возможности использования графических функций, в главе 23 мы создадим приложение для расчета наборов Мандельброта и создания удивительных по сложности и красоте рисунков в объекте холст.

Практикум

Вопросы и ответы

Для большинства приложений Windows характерна следующая схема отношений между окнами приложения. Одно большое окно программы содержит одно или несколько окон документов, не превышающих по размеру главное окно приложения. Возможно ли создание такой конструкции с графическими объектами Tkinter?

Речь идёт об интерфейсе MDI (Multiple Document Interface — многодокументный интерфейс). В Tkinter нет средств для поддержания подобного интерфейса. Почему бы Вам не заняться самим созданием класса такого интерфейса. Если бы я начинал что-либо подобное, то первое, что бы я сделал, это посетил домашнюю страницу Грега Мак-Фарлана. Я уже представлял Вам его в предыдущей главе. Напоминаю адрес: <http://www.dscpl.com.au/pmw/>.

Когда я запустил программу multi-editor.py на своём компьютере в системе Linux, то открылись ужасно старомодные диалоговые окна для открытия и сохранения файлов. Неужели я вынужден буду иметь с ними дело?

Совсем нет, разработайте свои диалоговые окна или дождитесь того момента, когда в библиотеку Tk/TCL будут добавлены средства KDE или GNOME. Если Вы хоть немного знакомы с программированием на C или C++, то для Вас не составит труда разработать окна, отвечающие вашему вкусу и потребностям. Начните с посещения домашней страницы KDE (K Desktop Environment) по адресу <http://kde.fnet.pl/> или с домашней страницы GNOME Developer (<http://developer.gnome.org/>).

Контрольные вопросы

1. В чем отличие между корневым окном и объектом окна верхнего уровня?
 - а) Корневое окно не может содержать графические объекты.
 - б) Объект окна верхнего уровня может содержать только некоторые графические объекты, непосредственно унаследованные от него.
 - в) Ничем.
 - г) Корневое окно не имеет родительских окон, а объект окна верхнего уровня является дочерним для корневого окна.
2. Почему в программе multi-editor.py для обработки событий используются методы?

- а) Функции не могут использоваться для обработки событий, только методы.
- б) Если бы использовались функции, мы бы никогда не узнали, из какого окна поступил вызов.
- в) Тем самым удалось существенно сократить число строк кода.
- г) Методы работают быстрее, чем функции.

Ответы

1. г. Во всех приложениях Tkinter существует только одно корневое окно, которое не имеет родительских окон. Объектов окон верхнего уровня может быть сколько угодно, но все они наследуются от одного корневого окна.
2. б. Методы принадлежат классам. Благодаря тому что определение поля редактора сделано в классе, метод обработки события всегда будет выполняться для того окна, из которого он был вызван.

Примеры и задания

В наши программы `tkeditor.py` и `multi-editor.py` прокралась одна ошибка. При сохранении файла в него может добавляться лишняя пустая строка. Найдите ошибку и устраните её.

Добавьте в наш текстовый редактор `tkeditor.py` новые средства и возможности. Например, сделайте следующее:

- добавьте возможность открытия редактора из командной строки с указанием в качестве аргумента имени файла, который нужно открыть для редактирования;
- добавьте возможность внедрять в текст графические изображения;
- модернизируйте функцию команды "Сохранить как..." таким образом, чтобы рисунки сохранялись вместе с файлом;
- добавьте в меню команду Сохранить;
- добавьте возможность выбора шрифта для текста;
- добавьте возможность изменять цвет шрифта;
- попробуйте преобразовать программу `tkeditor.py` или `multi-editor.py` в редактор документов HTML;
- добавьте документацию в программу редактора;
- представьте Ваше новое приложение на общедоступном ftp-сервере на суд общественности.

***Прим. В. Шипкова: указанные задания весьма полезно выполнить. После их выполнения станет весьма понятно на сколько легко работать с Питоном. Попробуйте нечто подобное сделать на Visual Basic или Visual C++. ;)**

20-й час

функции рисования библиотеки Tk, I

В предыдущих двух главах Вы познакомились с основными графическими объектами библиотеки Tkinter. В этой главе мы продолжим тему использования базовых графических объектов при создании интерфейса GUI и перейдем к работе с функциями рисования Tkinter. Изучив материал данной главы, Вы сможете использовать новые типы меню (всплывающие и динамические); объединять списки и полосы прокрутки в функциональные конструкции; добавлять в объекты Tk готовые графические изображения; различать графику разных типов в объекте холст.

Новые меню

В главе 18 Вы узнали об объекте меню, который является контейнером для объектов кнопок меню. Но мы познакомились лишь с двумя стандартными типами меню: строкой меню (которая располагается в верхней части окон большинства приложений) и с меню опций (может появляться в любом месте окна приложения). Прежде чем перейти к работе с графикой, нам нужно рассмотреть следующие дополнительные типы меню:

- всплывающее (или контекстное) меню, которое открывается после щелчка правой кнопки мыши;
- динамическое меню с обновляемым набором опций.

Всплывающие меню используются во многих приложениях. В листинге 20.1 показан пример кода такого меню.

Листинг 20.1. Программа tkrorup.py

```
from Tkinter import *
import string
import sys

def die(event=None):
    sys.exit(0)

def getxy(something):
    s=something.geometry()
    return map(int,
string.splitfields(s[1+string.find(s,"+"):],'+'))

def new_file(event=None):
    print "Opening new file"
```

```

def open_file(event=None):
    print "Opening existing file"

def activate_menu(event=None):
    x, y = getxy(root)
    menu.tk_popup(x+event.x, y+event.y)

root = Tk()
root.canvas = Canvas(root, height=100, width=100)
root.canvas.pack()
menu = Menu(root)
menu.add_command(label="New...", underline=0,
command=new_file)
menu.add_command(label="Open...", underline=0,
command=open_file)
menu.add_separator()
menu.add_command(label="Exit...", underline=0, command=die)
menu['tearoff'] = 0

root.canvas.bind("<Button-3>", activate_menu)

root.mainloop()

```

Пожалуй, только одна строка в этом листинге требует объяснения, поскольку со всеми остальными инструкциями кода мы уже встречались в предыдущих главах. Но выражение в строке 12 нарушает традиционную простоту кода и может вызвать у Вас затруднение. В строке 11 инструкция `s=something.geometry()` возвращает геометрические пропорции основного окна (`root`). Данный метод возвращает текстовую строку в формате `"100x100+64+64"`— стандартная запись, используемая в X Windows. Компонент `100x100` представляет ширину и высоту окна, в данном случае 100 на 100 пикселей. Компонент `+64+64` указывает координаты `x` и `y`, опять-таки, в пикселях, от левого и верхнего краев экрана. Чтобы оперировать этими данными, необходимо преобразовать части строки в соответствующие числовые значения. Для показа всплывающего меню, что происходит в строке 22, все, что нам нужно, — это координаты `x` и `y`. Функция `string.find()` отыскивает в возвращенной строке первое вхождение символа `+`. Мы прибавляем к полученному значению единицу (поскольку искомое значение следует за символом `+`), после чего вырезаем последовательность символов от цифры за знаком `+` и до конца строки. В нашем примере будет возвращена строка `64+64`. Затем мы преобразуем полученную строку в список строк, вновь используя символ `+` в качестве разделителя. Таким образом, функция `string.splitfields()` возвратит значение `[64, 64]`, которое передаётся в функцию `map()`.

Данная функция преобразовывает строки в соответствующие целочисленные значения с помощью вызова функции `int()`. И наконец, инструкция `return` принимает список целых чисел и возвращает его по месту вызова функции `getxy()`, где возвращённые значения присваиваются переменным `x` и `y`. К этим переменным прибавляются соответствующие значения `event.x` и `event.y`, в результате чего получаются экранные координаты точки щелчка мыши (нам нужны именно эти координаты, а не координаты окна `root`). Полученные значения используются в функции `menu.tk_popup()`.

Обычно во всплывающих меню не предлагаются такие опции, как "Закрыть...", так как во время работы со всплывающим меню гораздо проще допустить ошибку. Эта команда более характерна для основной строки меню, но в нашем простеньком примере, показанном на рис. 20.1, такая конструкция меню вполне допустима.

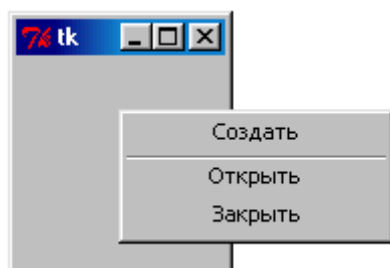


Рис. 20.1. Всплывающее меню, созданное программой `tkpopup.py`

Полезным средством является возможность динамического построения меню, т.е. вместо жёсткого указания в программном коде определённого набора опций можно дать задание программе конструировать меню в ходе выполнения, основываясь на предыдущих действиях пользователя. Написание функции для такого меню представляется достаточно сложной задачей, но в то же время это удобная тема для рассмотрения использования инструкции `lambda` (анонимной функции), которая, с моей точки зрения, в наибольшей степени подходит для программирования динамических меню. В листинге 20.2 показан пример программного кода динамического меню, вложенных меню и (как подарок) пример использования в приложениях предварительно созданных графических изображений.

Листинг 20.2. Программа `tkcascadingmenu.py`

```
from Tkinter import *
import os
import string
```

```

img=None

def die():
    sys.exit(0)

def listgifs(d="."):
    l=os.listdir(d)
    rl=[]
    for i in l:
        t=string.lower(i)
        g=string.rfind(t, ".gif")
        if g>=0:
            rl.append(i)
    if len(rl)<1:
        rl=None
    else:
        rl.sort()
    return rl

def setimage(s):
    global elements
    global lb
    global img
    if s in elements:
        img=PhotoImage(file=s)
        lb["image"]=img

def main():
    global elements
    global lb
    elements=listgifs()
    if not elements:
        print "No gifs"
    n=len(elements)
    nm=n/10
    no=n%10
    if no:
        nm=nm+1
    print "For %d files, I'll make %d menus" % ( n, nm )

    root=Tk()
    mb=Menu(root)
    cb=Menu(mb)
    cb.add_command(label="Exit",command=die)

    gm=Menu(mb)
    for i in range(nm):
        tm=Menu(gm)

```

```

    if i==nm-1 and no!=0:
        lim=no
    else:
        lim=10
    for j in range(lim):
        ne=(10*i)+j
        tm.add_command(label=elements[ne],
            command=lambda m=elements[ne]:setimage(m))
        gm.add_cascade(label="List gifs %d" % (i),menu=tm)

mb.add_cascade(label="File", menu=cb)
mb.add_cascade(label="Gifs", menu=gm)

lb=Label(root,text="No gif")
lb.pack()
root.config(menu=mb)
root.mainloop()

if __name__ == "__main__":
    main()

```

Функция `listgifs()`, показанная в строках 12-25, возвращает с помощью метода `os.listdir()` отсортированный список всех файлов GIF в текущем каталоге. Поскольку в Python отсутствует нечувствительная к регистру встроенная функция сравнения строк, нам пришлось создать собственную. С этой целью для каждого элемента списка, полученного с помощью функции `listdir()`, применяется метод `string.lower()`. Полученная строка из символов в нижнем регистре сравнивается с искомым суффиксом. В качестве альтернативы можно использовать модуль `fnmatch`, который предоставляет зависимые от платформы средства локализации файлов. Например, альтернативный код функции `listgifs()` показан в листинге 20.3.

Листинг 20.3. Использование модуля `fnmatch` в функции `listgifs()`

```

def listgifs(d="."):
    l=os.listdir(d)
    rl=[]
    for i in l:
        if fnmatch.fnmatch(i,"*.gif"):
            rl.append(i)
        if len(rl)<1:
            rl=None
    else:
        rl.sort()

```




```
return rl
```

Вторым параметром в функцию `f nmatch()` передаются джокерные, или подстановочные, символы, обычные для системы UNIX. Аналогичные подстановочные символы знакомы пользователям системы DOS. В табл. 20.1 описывается назначение различных подстановочных символов, используемых в модуле `fnmatch`.

Таблица 20.1. Подстановочные символы системы UNIX

Символ	Значение
*	Любой набор символов или их отсутствие
;	Любой одиночный символ
[abc]	Любые символы, заключённые в квадратные скобки
[a-z] или [0-9]	Любой одиночный символ, заключенный в заданном диапазоне
[!seq]	Любой одиночный символ, не указанный в seq

Формат. GIF (Graphic Interchange Format – формат обмена графическими данными) изначально разрабатывался в качестве формата графических файлов для использования только в CompuServe. Но в настоящее время – это основной формат графики в Internet. Полемика вокруг использования данного формата возникала в связи с непредсказуемой политикой компании Unisys – владельца патента на метод сжатия данных LZW (Lempel-Ziv-Welch), используемого для эффективного уменьшения размеров графических файлов. Далее на примерах Вы узнаете больше о формате GIF и особенностях его использования. Учтите только, что Tkinter не поддерживает использование метода `write()` для файлов GIF.

За использование файлов GIF в нашей программе отвечает функция `setimage()` – строки 26–32. Для считывания файлов и приведения изображений к формату, понятному для всех графических объектов Tkinter, используется встроенная функция `PhotoImage()`. Переменная `elements` хранит список файлов GIF, обнаруженных в текущем каталоге. Данный список возвращается функцией `listgifs()`. Переменная `lb` представляет объект ярлыка. Обратите внимание, что переменная `img`, которая представляет графическое изображение, должна быть глобальной. В противном случае после завершения выполнения функции `setimage()` переменная испарится и графический объект, которому было передано изображение, потеряет его.

В этой программе также продемонстрирован стиль программирования, альтернативный тому, которого мы придерживались раньше. Программы на языке C всегда содержат функцию `main()`, которая определяет точку начала выполнения программы при её запуске. В данном примере имитируется стиль языка C, хотя в действительности выполнение программы на языке Python принципиально отличается от языка C. В Python основная часть программы начинается за выражением `if __name__ == "__main__":`. В листинге, показанном выше, в этом блоке просто происходит вызов функции `main()`, что некоторым образом имитирует язык C. Я предпочитаю не использовать этот стиль хотя бы потому, что Python действительно сильно отличается от C или PASCAL, но поскольку подобная запись может повстречаться Вам на практике, следует уметь в ней разобраться.

В строках 37-50 создаётся список файлов GIF, который затем присваивается глобальной переменной, определяется число вложенных меню; после чего создаётся меню файл. В строках 51-63 показан традиционный подход к конструированию вложенных меню: создаётся объект меню, добавляется 10 имён файлов GIF в качестве опций меню, после чего вызывается функция `add_cascade()`, которая добавляет объекты кнопок меню в объект-контейнер. После завершения цикла в строках 64, 65 вся конструкция вложенных меню добавляется в строку меню. Любопытное решение представлено в строках 60, 61, где создаётся объект кнопки меню, а в качестве функции обработки события назначаем анонимную функцию `lambda`.

Нам нужно, чтобы при выборе пользователем опции меню открывался соответствующий список графических файлов. Было бы здорово, если бы объект кнопки меню был настолько умным, что мог отслеживать названия опций по мере их выбора. Но, к сожалению, этого делать он не умеет, следовательно, умными придётся быть нам. Мы не можем напрямую использовать функцию `setimage()`, так как в неё невозможно передать имя выбранной опции. Проблема с анонимной функцией `lambda` состоит в том, что в ней нельзя использовать оператор присвоения. Чтобы обойти эти ограничения, нам следует воспользоваться приёмом с аргументом, заданным по умолчанию. Рассмотрим отдельно выражение с инструкцией `lambda`:

```
lambda m=elements[ne]: setimage(m)
```

На момент создания подменю имя файла изображения уже должно быть известно, и оно определяется как `elements[ne]`. Для передачи этого имени в `lambda` используется инструкция `m=elements[ne]`, где переменной `m`



присваивается действительное имя файла. Затем эта переменная передаётся в функцию `setImage()`. Дальше все очень просто. На рис. 20.2 показан результат выполнения программы `tkcascadingmenu.py`.

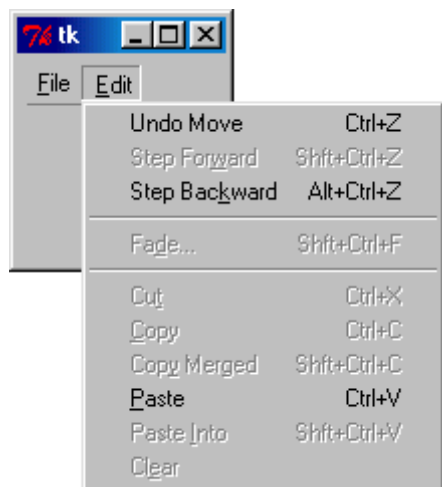


Рис. 20.2. Динамическое меню

Примите к сведению, что функция `PhotoImage()` различает только два графических формата: GIF и PPM. Пользователи Windows редко встречаются с файлами в формате PPM, но они обычны для систем Linux/UNIX. Поэтому нашу программу можно усовершенствовать, чтобы она отображала не только файлы GIF, но и PPM, если есть вероятность, что файлы этого формата могут быть представлены в каталогах предполагаемого пользователя.

Формат PPM (*Portable Pixmap* – переносимый массив пикселей) был разработан несколько лет назад Джефом Посканзером (*Jeff Poskanzer*) в качестве низкоуровневого общего формата для всех цветных графических изображений. Предполагалось, что этот формат заменит собой устаревший формат PBM (*Portable Bitmap* – переносимый массив битов), который поддерживал только черно-белые изображения. Данный формат не очень подходит для хранения изображений (хотя иногда используется в этих целях), поскольку он крайне не экономичен. Сохранение изображения в этом формате потребует значительно больше памяти по сравнению с традиционными графическими форматами. Чтобы ближе познакомиться с файлами PPM, обратитесь к разделу "Примеры и задания" в конце этой главы.

Любое меню может содержать подменю, причём нескольких уровней. Обычно создание многоуровневых меню – это не самое удачное решение по упрощению работы пользователя. Но иногда, при определённых обстоятельствах, создание подменю может оказаться весьма полезным, так как позволит сократить

число опций меню. Слишком длинный список опций может раздражать пользователя не меньше, чем необходимость блуждать по опциям многоуровневого меню. Необходимость сокращения числа опций меню становится тем более очевидной, если список опций не умещается на экране целиком. Также следует учитывать, что разрешение экрана пользователя может быть меньше, чем у разработчика. Например, многие разработчики приложений устанавливают разрешение экрана 1280x1024. (Один мой друг предпочитает разрешение 1600x1280. Он говорит, что при таком разрешении со стороны трудно определить, работает он или спит за компьютером. Если бы существовала возможность установить ещё большее разрешение, не сомневаюсь, что он воспользовался бы этой возможностью.) В то же время у пользователя может быть установлено разрешение экрана 640x480, в результате чего меню, великолепно выглядевшие на экране разработчика, могут вылезти за экран пользователя.

Прокручиваемые списки

Прокручиваемые списки следует отличать от комбинированных списков, которые встречаются во многих приложениях Windows. Комбинированные списки представляют собой однострочные редактируемые поля с кнопками, открывающими список опций. В отличие от них, прокручиваемые списки обычно представлены многострочными полями фиксированного размера, содержимое которых, можно прокручивать с помощью полосы прокрутки. Базовые графические объекты списка и полосы прокрутки имеют все необходимые встроенные функции, так что нам останется только объединить их в одно целое. В листинге 20.4 показано, как это сделать.

Листинг 20.4. Программа tkscrolledlistbox.py

```
from Tkinter import *
import sys
import os
import string

def setn(event):
    global elements
    global listbox
    global lb
    global img
    x=listbox.curselection()
    n=string.atoi(x[0])
    img=PhotoImage(file=elements[n])
    lb["image"]=img
```

```

def listgifs(d=".") :
    l=os.listdir(d)
    rl=[]
    for i in l:
        t=string.lower(i)
        g=string.rfind(t, ".gif")
        if g>=0:
            rl.append(i)
    if len(rl)<1:
        rl=None
    else:
        rl.sort()
    return rl

def die(event):
    sys.exit(0)

root=Tk()
button=Button(root)
button["text"]="Quit"
button.bind("<Button>",die)
button.pack()
labelx=Label(root)
labelx["height"] = 1
labelx.pack()

elements=listgifs()

frame=Frame(root,bd=2,relief=SUNKEN)
frame.pack(expand=1,fill=BOTH)
scrollbar=Scrollbar(frame, orient=VERTICAL)
listbox=Listbox(frame,exportselection=0,height=10,
yscrollcommand=scrollbar.set)
listbox.bind("<Double-Button-1>",setn)
for i in elements:
    listbox.insert(END, i)
scrollbar.config(command=listbox.yview)
listbox.pack(side=LEFT)
scrollbar.pack(side=LEFT, fill=Y)
listbox.select_set(0)
listbox.see(0)
lb=Label(root,text="No gif")
lb.pack()

root.mainloop()

```

В основу кода программы tkcascadingraenu.py были положены многие средства, с которыми мы уже познакомились в предыдущей программе. Функция listgifs() осталась той же, а

функция `setn()` напоминает `setimage()`. Поскольку в этой программе мы обошлись без анонимной функции `lambda`, функцию `setn()` можно было вызывать напрямую. В строке 51 мы связали вызов этой функции с событием двойного щелчка на опции списка, в результате чего загружается новое изображение. Новые инструкции представлены в строках 49-59, где создается объект списка `listbox`. Свойству `yscrollcommand` этого объекта присваивается метод `scrollbar.set()`. В свою очередь, в строке 54 свойству `command` объекта полосы прокрутки `scrollbar` присваивается встроенный метод списка `yview()`. Этих двух условий достаточно для того, чтобы два базовых графических объекта, список и полоса прокрутки, работали) как одно целое. В строке 57 устанавливается выбор первого элемента списка по умолчанию, а в строке 58 – показ выбранного изображения на экране. Чтобы загрузить другое изображение, достаточно дважды щелкнуть мышью на имени файла в списке, как показано на рис. 20.3.

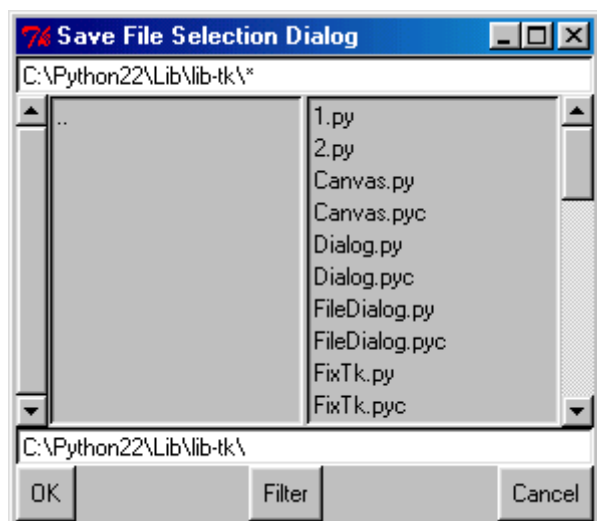


Рис. 20.3. Выполнение программы `tkcascadingmenu.py`

Если Вы модифицировали предыдущую программу `tkcascadingmenu.py` для показа файлов PPM, то сейчас можете просто перенести и вставить измененный код функции `listgifs()` в программу `tkcascadingmenu.py`.

Библиотека средств поддержания графики в Python

Если Вы хотите использовать дополнительные форматы графических файлов, загрузите с Web-страницы (<http://www.pythonware.com>) и установите на свой компьютер приложение Python Imaging Library. Ниже приведена инструкция по инсталляции программы в Windows.

1. Загрузите программный пакет, находящийся по адресу <http://www.pythonware.com/>.

2. Разархивируйте содержимое в папку Temp. В этой папке Вы затем обнаружите новую папку с вложенными папками PIL, Scripts и файлом tkinter.dll.
3. Скопируйте все файлы из папки PIL в папку Python\Lib на Вашем компьютере (например, в C:\Python\Lib).
4. Скопируйте все файлы из папки Scripts в папку Python\Tools\Scripts.
5. Скопируйте файл tkinter.dll в папку Python.

После выполнения этой процедуры Вы получите возможность использовать другие графические форматы. Для установки программы на машинах с Linux или UNIX необходимо скомпилировать файлы в соответствии с инструкцией, прилагаемой к установочному пакету (в том же каталоге на ftp-сервере). После установки использование программы не представляет труда, как Вы в этом убедитесь на приведённых ниже примерах. Только в дополнение к модулю Tkinter в программу следует импортировать ещё два модуля: `import Image, ImageTk`. При таком импортировании вызовы функций `img=PhotoImage(...)` следует заменить на `img=ImageTk.PhotoImage(...)`. Теперь Вы получили доступ к дополнительным графическим файлам. Пример такой программы приведён в листинге 20.5.

Листинг 20.5. Программа tkpil.py

***Прим. В. Шипкова: этот файл всё равно работать не будет без указанной библиотеки, поэтому код программы не привожу, кому надо можете [скачать файл](#).**

Почти всё в этой программе Вам уже знакомо. Код создания меню был перенесен в функцию `buildmenu()`. Для смены текущего каталога добавлены диалоговое окно открытия файла (функция `ls()` в строке 115) и ещё одно небольшое диалоговое окно, определённое в строках 47–67. Последнее диалоговое окно вызывается функцией `cd()` в строке 69, но все его свойства задаются в собственном коде, а именно в функции `ok()` в строках 57–67. Основная нагрузка ложится на строки 62–67, где происходит вызов функции `os.chdir()`, активизируется новый каталог, создаётся список файлов и т.д. Диалоговое окно для ввода пути к новому каталогу показано на рис. 20.4.

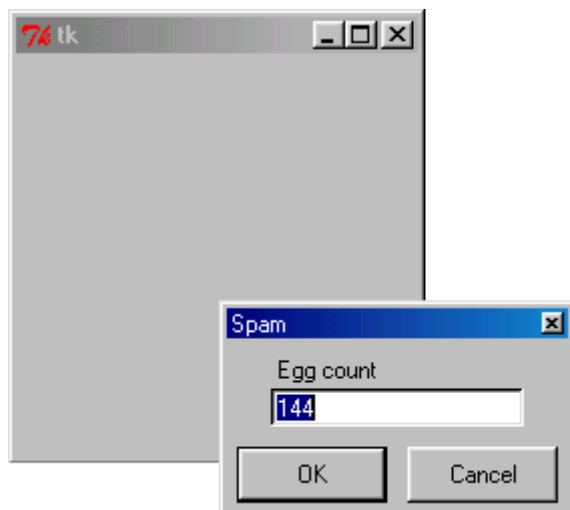


Рис. 20.4. Диалоговое окно Введите новый каталог приложения tkpil.py

В меню "Файл" была добавлена опция "Сохранить как...", с помощью которой любое открытое изображение можно сохранить в формате PPM. (Класс ImageTk предлагает для записи только этот формат. Другие форматы представлены в библиотеке PIL, но в этом случае следует использовать альтернативные пути загрузки файлов, описанные в документации, прилагаемой к PIL.) На рис. 20.5 показано диалоговое окно, открываемое функцией askopenfilename(). Аналогичное окно открывается также функцией asksaveasfilename().

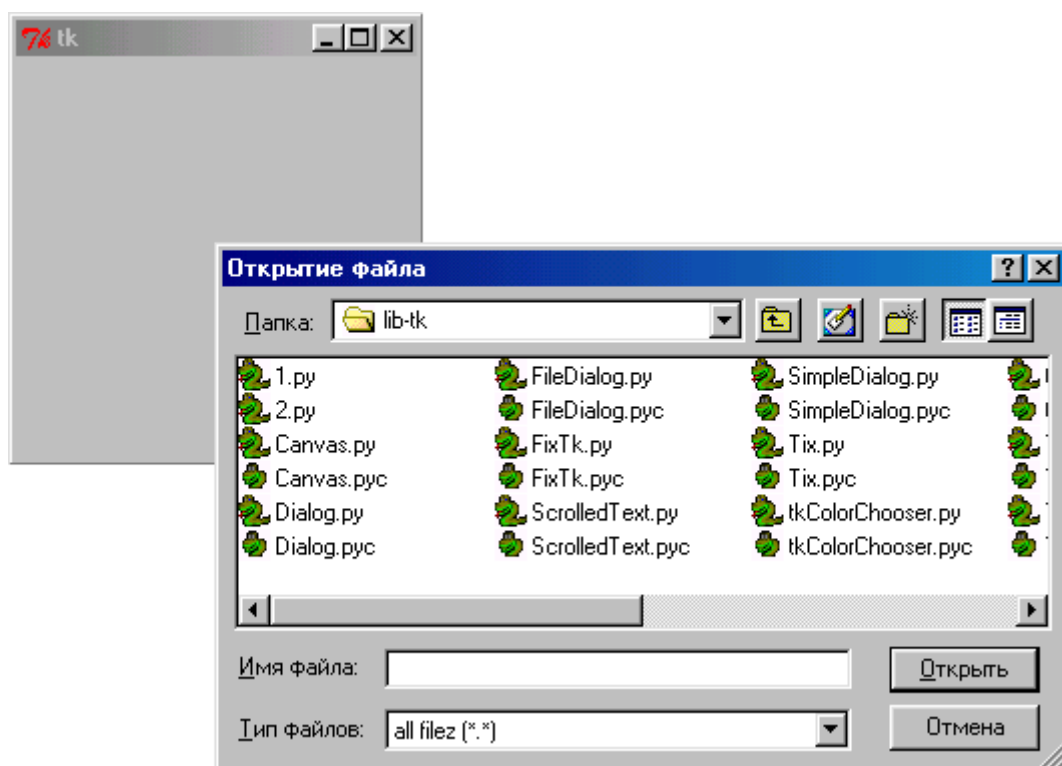


Рис. 20.5. Диалоговое окно Открытие файла приложения tkpil.py

***Прим. В. Шипкова: Это окно внешне отличается в Windows XP. Причина тому – изменившаяся системная библиотека, которая отвечает за прорисовку этого окна (хотя называется оно также).**

Если Вы всерьёз хотите заняться созданием графического редактора, то установка Python Imaging Library не окажется для Вас пустой тратой времени. С помощью средств этой библиотеки Вы сможете создавать собственные графические изображения и сохранять их в разных форматах, а не только в GIF. Хотя не надейтесь, что все проблемы решатся сами собой. В частности, некоторые сложности возникнут с использованием графических форматов XBM и XPM, распространенных в среде UNIX. Но это уже детали, требующие отдельного разбирательства.

В следующем разделе мы начнём работу над созданием графических изображений и рисунков с помощью объекта холст.

Графический объект холст

Графический объект холст библиотеки Tkinter создан специально для работы с графикой и содержит множество разнообразных методов. Документация с описанием методов разных графических объектов представлена на Web-странице по адресу <http://www.python.com/>, но она все ещё не завершена и несколько фрагментарна. Прежде чем приступить к использованию холста, следует уяснить некоторые общие принципы процесса рисования. Большинство инструментов рисования обычно не применяется для настройки цвета отдельных пикселей. Вместо этого создаётся некоторая структура данных, которая с помощью определённой функции передаётся окну или графическому объекту, внутренний код которого выполняет за Вас всю остальную работу. Если раньше Вам приходилось работать с другими инструментами рисования, то набор методов объекта холст может показаться довольно бедным. Тем не менее путём комбинирования имеющихся базовых функций можно создать практически любое изображение. Например, холст не содержит команды установки цвета отдельного пикселя, но это задачу можно выполнить с помощью метода `Rectangle()` или `Line()`. Мы хорошо попрактикуемся с комбинированием базовых методов рисования при создании приложения Mandelbrotset в главе 23.

Принципы, положенные в основу рисования в холсте Tkinter, несколько отличаются от принципов вывода изображений в других объектах GUI. В большинстве систем графического редактирования для рисования на экране можно использовать точки, линии и другие графические примитивы, причём

результаты рисования тут же появятся на экране. В случае с холстом Tkinter происходит предварительное описание графического элемента, такого, например, как прямоугольник, после чего он помещается в холст в список элементов, которые следует нарисовать. Элементы в списке не появятся на экране до тех пор, пока не поступит команда на обновление холста. После выполнения обновления все элементы из списка появятся на экране поверх ранее нарисованных элементов. Также можно дать команду на удаление нарисованного элемента, и он исчезнет после следующего обновления. Независимо от сложности нарисованного элемента, его можно легко перемещать по экрану с помощью простых команд. Для упрощения работы пользователей можно запрограммировать выделение на экране наборов элементов, которые будут перемещены или удалены. Позже, на примере графического редактора Mandelbrot, мы рассмотрим на практике, как все это делается. Недостаток данного подхода рисования состоит в том, что объект холста сохраняет в себе все созданные графические элементы, число которых может превышать тысячу. Для редакторов растровой графики, где учитывается только цвет отдельных пикселей, усложнение рисунка не влияет на его размер. Напротив, с добавлением каждого нового элемента объект холста требует все больше памяти, а время его обновления катастрофически возрастает. Впрочем, ресурсов современных компьютеров обычно вполне хватает на то, чтобы удовлетворить запросы в графике большинства пользователей.

В следующих разделах будут описаны основные графические элементы, поддерживаемые объектом холст библиотеки Tkinter. В каждом случае также будут рассматриваться специальные методы, используемые в этом объекте для создания и редактирования данных элементов. По возможности всегда используйте встроенные методы, хотя можно создавать собственные, основываясь на базовых методах объекта холст: `create_rectangle()` и `create_line()`.

Учтите, что для использования специальных методов их следует импортировать из модуля Canvas.

Прямоугольник

Прямоугольники задаются четырьмя координатами: x - y — для верхнего левого угла и x_1 - y_1 — для нижнего правого угла. Нулевые координаты x - y соответствуют верхнему левому углу холста. Но чтобы пиксель с координатами 0,0 был виден на экране, нужно обнулить свойства холста `borderwidth` и `highlightthickness`. Если этого не сделать, то рамка текущего фокуса будет прочерчена по верхнему краю холста и

скроет часть крайних пикселей. Чтобы очертить рамку вокруг холста, лучше всего вставить объект холста в объект рамки и установить для него соответствующее свойство границы.

Синтаксис вызова метода `Rectangle()` для рисования прямоугольника следующий:

```
Rectangle (cv, x, y, x1, y1, опции...)
```

Например: `Rectangle(cv, 100, 100, 200, 200, fill="red")`

В результате в объекте холста `cv` будет нарисован квадрат со стороной, равной 100 пикселям, залитый красным цветом с координатами верхнего левого угла 100 пикселей от верхнего края холста и 100 пикселей от левого края (в терминологии Tk этот угол называется NW – *northwest* – северо-западный). Ту же операцию можно выполнить с помощью следующего базового метода:

```
cv.create_rectangle(x, y, x1, y1, опции...)
```

Вместо того чтобы перечислять здесь все опции, среди которых много весьма полезных, лучше загрузите из Internet демонстрационную программу, представленную по адресу <http://www.pauahtun.org/TYPython/CanvasDemo.html>. Прилагаемые инструкции и экранные подсказки помогут Вам разобраться со всеми опциями графических элементов, поддерживаемых объектом холст библиотеки Tkinter.

Линия

Линии, как и прямоугольники, задаются с помощью четырех координат. Координаты `x` и `y` указывают исходную точку линии, а `x1` и `y1` – конечную точку. Используется следующий синтаксис:

```
Line (cv, x, y, x1, y1, опции...).
```

Например:

```
Line(cv,100,100,200,100,fill="blue").
```

Дуга

Следует отличать дуги, создаваемые с помощью метода `Arc`, от окружностей, для создания которых используется метод `Oval` (подробно об этом – в следующем разделе). На рис. 20.6 показана копия экрана демонстрационной программы `CanvasDemo.py`, которую можно загрузить из Internet. Дуги в верхней части рисунка называются секторами, в среднем ряду

— собственно дугами, а в нижнем ряду — хордами. Так выглядят вызовы методов для построения дуг всех трёх типов:

```
Arc(cv, x, y, x1, y1, extent=90, fill="#808080",\
    style="pieslice")
Arc(cv, x, y, x1, y1, extent=90, fill="# 808080",\
    style="arc")
Arc(cv, x, y, x1, y1, extent=90, fill="#1808080",\
    style="chord")
```

Как видите, различные типы дуг задаются только параметрами. С помощью метода Arc также можно построить замкнутую окружность, но гораздо эффективнее для этого воспользоваться методом Oval.



Рис. 20.6. Создание дуг с помощью метода Arc

Окружность и овал

Чтобы нарисовать окружность, такую же, какую в предыдущем разделе мы создали с помощью метода Arc, гораздо проще воспользоваться следующим методом:

```
Oval(cv, x, y, x1, y1, fill="#808080")
```

Следует помнить, что ни окружность, ни сектор и ни овал не рисуются от центра (к тому же в овале два фокальных центра). Все эти фигуры задаются прямоугольником, в который они вписаны. Центр фигуры не указывается в параметрах. Его можно только рассчитать, если в этом возникнет необходимость. Оперировать прямоугольником значительно проще. Так, на рис. 20.7 показано, как можно из совершенной окружности получить совершенный эллипс, изменяя стороны прямоугольника, вписанные в этот прямоугольник.

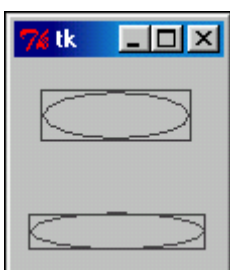




Рис. 20.7. Изменяя стороны прямоугольника, можно легко менять форму вписанного в него эллипса

Многоугольник

Многоугольником называется закрытая фигура, построенная из линейных сегментов. Можно использовать неограниченное число сегментов, но не меньше трёх. Невозможно построить замкнутую фигуру, содержащую меньше трёх сторон. Многоугольники со сторонами равной длины и равными углами называются *правильными*. Для любого числа сегментов можно создать правильный многоугольник: треугольник, квадрат, пятиугольник и т.д. Правильный многоугольник, состоящий из огромного числа граней и углов, будет не отличим от окружности. Это возможный путь создания окружности, но не самый эффективный. Гораздо проще воспользоваться методом `Oval`. Простейшим многоугольником является треугольник. На практике гораздо чаще используются неправильные треугольники, чем правильные, что, впрочем, справедливо и для других многоугольников. Метод `Polygon` можно использовать для рисования как правильных, так и неправильных многоугольников. Но если Вам нужно построить правильный многоугольник, следует воспользоваться программой расчёта координат вершин правильной фигуры. Ряд таких программ можно бесплатно загрузить из Internet, но среди них нет написанных на Python. (Дополнительную информацию о построении правильных фигур Вы найдёте в разделе "Примеры и задания" в конце главы.)

Вершиной фигуры называется точка, где сходятся две её грани. Правда, если две грани сходятся под углом 180° , то вершина не получится. Это будет просто сплошная линия.

В случае с методом `Polygon` объект холст не выполняет за Вас никаких вычислений, как это было при рисовании окружностей, дуг и эллипсов. В объект просто передаётся список координат (x, y) , и программа прочерчивает линии между этими вершинами. Если нужно построить девятиугольник, то в коде следует указать координаты всех девяти вершин, но, в отличие от многих других графических систем, Вам не нужно в конце списка повторно указывать первую вершину, чтобы замкнуть фигуру. Программа Tkinter автоматически прочерчивает линию между первой и последней вершинами. Ниже показан код создания четырехугольника:

```
Polygon(cv, 130, 130, 130, 258, 258, 258, 130, 30, fill="#FFFFFF")
```

В этом примере предполагается, что объект холст имеет размер 400x400 пикселей с цветом фона, отличным от белого

(обычно оттенки серого) . Подробнее о рисовании многоугольников мы поговорим позже .

Рисунок

Побитовые изображения называют ещё *однослойными* . Для поддержания цветовой информации требуется несколько слоев , поэтому однослойные побитовые изображения всегда монохромные . Рисунок создаётся из мозаики пикселей , которые могут быть только белыми или черными (включен/выключен) . Обычным форматом для сохранения таких изображений является широко используемый в X Windows формат XBM - X Bitmap . Как Вы уже видели раньше , библиотека Tkinter поддерживает использование изображений такого типа с помощью объекта ImageTk . При работе с побитовыми изображениями можно начинать не с создания самого изображения , а с объекта , в котором следует определить файл источника и установить цвета переднего и заднего плана . Пример такого кода показан ниже ,

```
bmpn = "@GVI.xbm"  
item =  
Bitmap(cv, 368, 368, bitmap=bmpn, foreground="#808080")
```

Символ @ указывает , что функции Bitmap() следует искать файл , а не встроенное побитовое изображение . Вновь предполагается , что объект холста имеет размер 400x400 пикселей . Результат выполнения этих инструкций показан на рис. 20.8 .



Рис. 20.8. Побитовый рисунок

Изображение

Метод ImageItem используется так же , как и метод Bitmap , но для представления не только черно-белых , но и цветных рисунков . Метод PhotoImage поддерживает изображения только в форматах GIF и PPM . Как Вы узнали в этой главе , для использования других форматов необходимо установить библиотеку PIL . Следующий код отобразит цветное изображение в том же холсте , что и предыдущее побитовое изображение :

```
img=PhotoImage(file="GVI.ppm")
item=ImageItem(cv, 368, 368, image=img)
```

На рис. 20.9 показано полученное изображение.



Рис. 20.9. Графическое изображение Image

Текст

Для добавления в холст текста используется метод CanvasText. В параметрах метода нужно задать координаты текста, сам текст и цвет шрифта. Ниже представлен пример вывода текста на экран в объекте холста,

```
item =
CanvasText(cv, 368, 368+48, fill="#007090", text="Бог
индейцев Майя")
```

Результат показан на рис. 20.10.



Рис. 20.10. Текст и графическое изображение, добавленные в холст

Обратите внимание, что по умолчанию надпись выравнивается по центру относительно заданных координат. Чтобы выровнять надпись по левому краю, следует установить параметр anchor=NW.

Метод Window

Метод Window используется для того, чтобы разместить различные графические объекты Tkinter внутри холста. В следующем коде показан пример добавления в холст кнопки:

```
btn=Button(cv, text="Показать рисунок", command=drawImage)  
item=Window(cv, 368, 368, anchor=SW, window=btn)
```

На рис. 20.11 показан холст, содержащий кнопку.

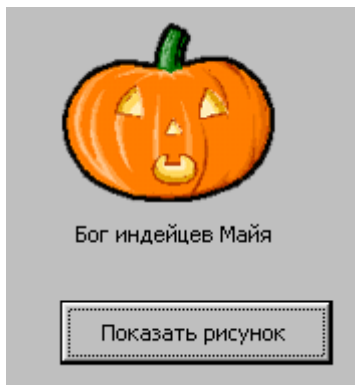


Рис. 20.11. Кнопка, добавленная с помощью метода Window (это указывает на тот факт, что все кнопки на экране также являются окнами)

Резюме

Мы рассмотрели разные типы меню, научились создавать всплывающие меню и списки с полосой прокрутки, узнали о базовых средствах рисования пакета Tkinter (включая Python Imaging Library), получили общие представления о методах редактирования изображений в объекте холста и обсудили различные базовые графические элементы, которые можно использовать в холсте.

В следующей главе мы рассмотрим работу с простыми изображениями и возможности анимации рисунков.

Практикум

Вопросы и ответы

Можно ли создавать игры с помощью Python?

Безусловно. Правда, если весь код написать только с помощью Python, программа будет не очень быстрой. Хотя бы основные блоки следует создать с помощью языка C. На 7-й международной конференции по языку Python, которая проходила в Хьюстоне (Houston) в 1998 г., был представлен имитатор Virtual World, полностью написанный на языке Python. Более подробную информацию о нем Вы найдёте на Web-странице по адресу <http://www.foretec.com/python/workshops/1998-11/proceedings/papers/asbahr/asbahr.html>.

***Прим. В. Шипкова: на самом деле на конец 2005 года существует по крайней мере один пакет для профессиональной работы с 3D-графикой, один пакет для создания игр, и плагин написанный на Python для одной весьма популярной программы для работы с 3D-графикой.**

На рис. 20.6 представлено много интересных графических объектов. Почему мы не рассмотрели их подробно?

Это объекты Python MegaWidgets, рассмотрение которых выходит за рамки данной книги. Обратитесь к Web-странице PMW по адресу <http://www.dscpl.com.au/pmw/>.

Контрольные вопросы

1. Как добавить в список полосу прокрутки?
 - а) Нужно разработать отдельные функции для полосы прокрутки и для списка.
 - б) Нужно создать только функцию для полосы прокрутки.
 - в) Необходимо сконструировать для списка собственную полосу прокрутки.
 - г) Можно просто воспользоваться встроенными функциями объектов полосы прокрутки и списка.
2. Какая разница между методами Bitmap и ImageItem?
 - а) Нет никакой принципиальной разницы.
 - б) С помощью метода ImageItem можно добавлять цветные изображения, а с помощью Bitmap – только монохромные.
 - в) Это два альтернативных метода использования рисунков в приложениях,
 - г) Методы отличаются только скоростью выполнения.

Ответы

1. г. Все функции, необходимые для прокручивания списков, уже встроены в объекты списка и полосы прокрутки. Нужно только присвоить их соответствующим параметрам.
2. б. Действительно, метод ImageItem отличается только способностью обрабатывать цветные изображения. Каждый пиксель в объекте Bitmap представлен только одним битом данных, тогда как для сохранения информации о цвете пикселя в объекте Image требуется по крайней мере 8 бит.

Примеры и задания

Дополнительную информацию об использовании анонимных функций можно получить на домашней странице International Obfuscated C Code Contest (Международный диспут по неявному программированию на C) по адресу <http://www.ioccc.org/>. И хотя основное внимание уделено программированию на языке C, в действительности неявные (анонимные) коды можно создавать на любом языке. Python, пожалуй, меньше всего подходит для этих целей, но и ему по силам создавать подобные коды для чётко детерминированных несложных приложений.

Больше сведений о графическом формате GIF и связанных с ним проблемах можно найти на Web-странице Burn All Gifs (Первоисточник файлов GIF) по адресу <http://burnallgifs.org/>. Эрик Раймонд (Eric S. Raymond) опубликовал ряд интересных статей по этой теме на своей Web-странице, её адрес – <http://www.tuxedo.org/~esr/>.

За информацией о netpbm и других графических форматах зайдите на страницу Graphics Muse (Муза графики) по адресу <http://www.graphics-muse.org/>.

Совсем недавно Тим Миддлетон (Tim Middleton) открыл новую страницу на сервере Vaults of Parnassus по адресу <http://www.vex.net/parnassus/>. Было заявлено, что на этой странице будет собрана одна из лучших подборок программных модулей на Python.

***Прим. В. Шипкова: Товарищ, надо признать, постарался. Хотя на этом сайте и не самые новые версии программ, но общее представление о той нише, которой занимается Питон в мире программирования – получить можно. ;)**

Действительно, здесь можно найти отличные программные продукты, которые великолепно иллюстрируют применение на практике всех средств программирования, рассмотренных в этой книге. Данная страница задумывалась как эквивалент архивов CPAN, где собраны стандартные модули на языке Perl. (Между прочим, Парнассус (Parnassus) – это гора, в расщелинах которой, согласно древнегреческой мифологии, скрывался Питон.)

Чтобы научиться создавать правильные многоугольники, посетите Web-страницу по адресу <http://www.scienceu.com/geometry/articles/tiling/>.

21-й час

В этой главе мы продолжим изучение функций рисования библиотеки Tk и рассмотрим средства управления цветом и создания простейшей анимации. Изучив материал данной главы, Вы получите общие представления о программировании цвета и сможете добавлять в объект холста простейшую анимацию.

Цвет

На тему управления цветом в программах написаны сотни, если не тысячи книг. Для тех, кто хочет постичь основы это вопроса, можно порекомендовать начать с книги Чарльза Поинтона (Charles Poynton) *Digital Video*.

Мы лишь в общих чертах рассмотрим некоторые общие принципы управления цветом, так как для более детального их рассмотрения пришлось бы написать отдельную книгу. Если же Вы захотите профессионально овладеть этим мастерством, то Вам потребуются годы практики (по крайней мере не 24 часа).

Приготовьтесь к стремительному броску к истокам программирования цвета. В 1931 г. в CIE (Le Commission Internationale de L'Eclairage – Международная комиссия по вопросам освещения) была разработана математическая модель цвета, предложенная в качестве международного стандарта. Основываясь на том, что глаза человека воспринимают цвет фоторецепторами трёх типов (если Вы помните из курса биологии средней школы, эти фоторецепторы называются колбочками), в модели CIE было предложено изображать все цвета как точки в трёхмерном пространстве. Таким способом можно представить в цифровом выражении любой видимый цвет. Далее мы ограничимся рассмотрением аналогичной цветовой схемы, называемой RGB (red-green-blue – красный-зеленый-синий). Именно эта схема наиболее часто используется в компьютерной графике. Схема RGB линейна, т.е. все три оси равнозначны и разделены на единичные отрезки равной длины. Глаза человека по-разному воспринимают изменение интенсивности освещения – в зависимости от уровня освещенности и длины световой волны. Именно поэтому не все цвета, видимые глазом, можно корректно воспроизвести в модели RGB. Только модель CIE обладает достаточной сложностью, чтобы смоделировать цветное зрение человека. Все остальные модели были получены как компромисс между требованием к точности цветопередачи и экономичностью модели. Учитывалась также область применения модели. Например, для телевидения была разработана и используется специальная цветовая модель YIE. Но она не используется в компьютерной графике, поэтому мы и не рассматриваем её в этой книге. Графическое представление модели RGB показано на рис. 21.1. К сожалению, на рисунке модель стала черно-

белой с оттенками серого. Чтобы оценить всю гамму красок, запустите на своём компьютере программу `scube.py`.

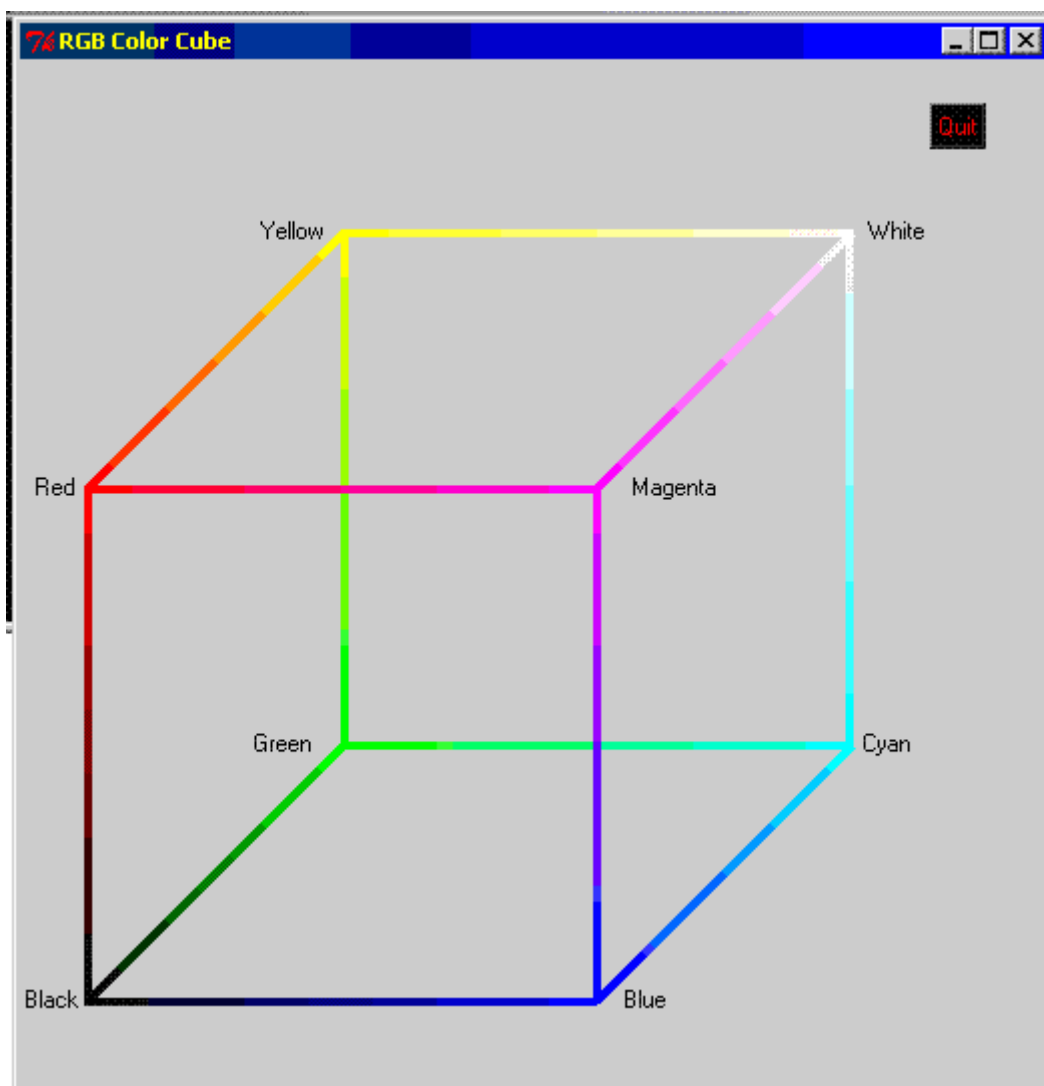


Рис. 21.1. Графическое представление цветовой схемы RGB

***Прим. В. Шипкова: в оригинале рисунок был другой, но поскольку он близко не похож, счёл уместным его заменить.**

В цветном изображении на экране компьютера Вы увидите, как чёрный цвет плавно переходит в красный по переднему левому вертикальному ребру куба, зеленый переходит в желтый по заднему левому вертикальному ребру и т.д. Код программы `scube.py` показан в листинге 21.1.

Листинг 21.1. Программа `scube.py`

***Прим. В. Шипкова: Ещё один длинный листинг, который лень переделывать. Кому надо – [брать здесь](#). Длина листинга более 270 строк.**

(Если цветовая палитра дисплея – 256 цветов, то значение `ncolors` следует изменить на 16, иначе цвета в кубе будут отображаться неправильно.) Большая часть кода должна быть Вам хорошо знакома по предыдущим главам, в которых мы изучали различные графические объекты библиотеки Tk. Так, в программе Вам повстречались уже знакомые функции рисования прямоугольников и многоугольников в объекте Canvas (холст). В строках 221-223 создаётся кнопка Выход, которая добавляется в холст с помощью функции `Window()`. Единственное, что Вам до сих пор не встречалось, – это повторяющиеся выражения, первое из которых появляется в строке 11:

```
cm = "%02X%02X%02X"%(brl[0], brl[1], brl[2]).
```

В результате выполнения этого выражения получается строка, которая может выглядеть как "1000000" для черного цвета или "FFFFFFF" – для белого. Для Tkinter цвета должны задаваться либо ключевыми словами, такими как "red", "white" или "black", либо строковыми значениями цветов модели RGB, которые начинаются с символа #. Следующие за этим символом шесть цифр определяют координаты цвета в трехмерном пространстве RGB. Первые две цифры представляют красную составляющую цвета, вторая пара цифр – зеленую, а третья пара – синюю составляющую. Таким образом, тёмно-красный цвет можно задать значением "#800000" – половинное значение яркости красной составляющей при полном отсутствии зеленой и синей. В функциях листинга 21.1 все стороны куба подразделены на 32 сегмента (число градаций цвета). Каждый сегмент, представляющий свой цвет, прорисовывается с помощью функции `Rectangle()` или `Polygon()`. Цветовые диапазоны для всех сторон куба показаны в табл. 21.1 (в том же порядке, в каком они создаются).

Таблица 21.1. Цветовые диапазоны сторон куба

Сторона куба	Цветовой диапазон	Функция
От зеленого к синему (задняя нижняя, слева направо)	#00FF00-#00FFFF	GreenCyan()
От зеленого к желтому (задняя слева, снизу вверх)	#00FF00-#FFFF00	GreenYellow()
От желтого к белому (задняя верхняя, слева направо)	#FFFF00-#FFFFFF	YellowWhite()
От белого к синему (задняя правая, снизу вверх)	#FFFFFF-#00FFFF	WhiteCyan()
От черного к голубому	#000000-#0000FF	BlackBlue()

(передняя нижняя, слева направо)		
От черного к красному (передняя слева, снизу вверх)	#000000-#FF0000	BlackRed()
От красного к пурпурному (передняя верхняя, слева направо)	#FF0000-#FF00FF	RedMagenta()
От пурпурного к голубому (передняя правая, снизу вверх)	#FF00FF-#0000FF	MagentaBlue()
От черного к зеленому (нижняя левая, спереди назад)	#000000-#00FF00	BlackGreen()
От голубого к синему (нижняя правая, спереди назад)	#0000FF-#00FFFFFF	BlueCyan()
От фиолетового к желтому (верхняя левая, спереди назад)	#FF0000-#FFFF00	RedYellow()
От пурпурного к белому (верхняя правая, спереди назад)	#FF00FF-#FFFFFF	MagentaWhite()

В результате подразделения цветовых диапазонов на 32 градации создается палитра, состоящая из 32768 цветов (32^3). Мы могли бы произвольно увеличить количество цветов в палитре, увеличив число градаций. Единственным ограничением в этом случае является максимально допустимое в Python значение числа с плавающей запятой. Представьте себе куб RGB как хранилище денег дядюшки Скруджа, где каждая монета или банкнот описывается уникальными координатами в трехмерном пространстве, в результате чего имеет уникальный цвет. Максимальная возможность хранилища вместить как можно больше денег зависит только от размеров монет или банкнот (по мультфильму помню, что дядюшка Скрудж предпочитал монеты). Видеокарты большинства компьютеров позволяют оперировать цветовыми палитрами от 16 до 16777216 цветов. В последнем случае стороны куба подразделяются на 256 цветовых градаций — максимальный набор, который можно представить с помощью строковых цветовых значений библиотеки Tkinter.

Цветной куб RGB наглядно иллюстрирует геометрический принцип оцифровки цвета, но на этой модели плохо видны переходы между цветами. Давайте рассмотрим другую модель, известную Вам со школьных уроков по физике, — цветное колесо. Пример такого колеса показан на рис. 21.2. Часто

цветное колесо состоит из сегментов смежных цветов, но в нашем случае задан плавный градиентный переход между цветами.

***Прим. В. Шипкова: учебник несколько устарел. Питон свободно оперирует цветами, и рисунок, который был приведён в оригинале пришлось заменить.**

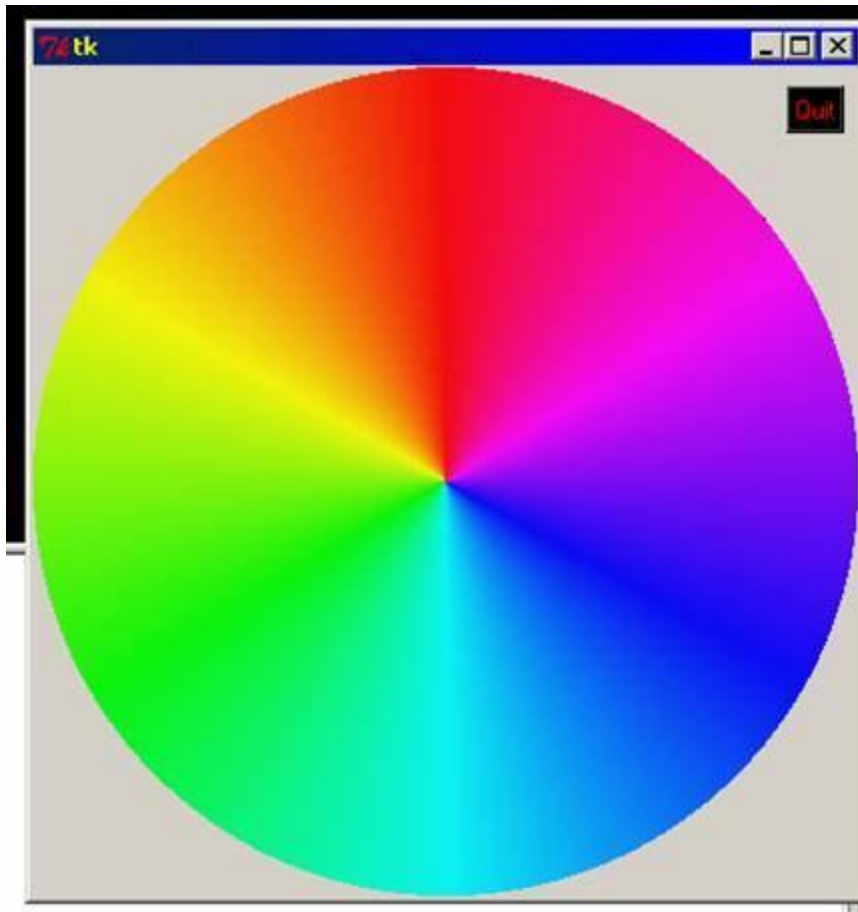


Рис. 21.2. Цветное колесо

Верхняя часть колеса красная. Под углом 120° вправо представлен сектор зеленого цвета, а под углом 120° влево – синего цвета. Это основные цвета модели RGB. Голубой сектор находится на противоположной стороне колеса от красного, пурпурный противостоит зеленому, а желтый – синему. Эти цвета называются *комплементарными*. Чтобы получить цветное колесо на экране своего компьютера, выполните программу `tkwheel.py`, код которой представлен в листинге 21.2.

Листинг 21.2. Программа `tkwheel.py`

```
from Tkinter import *
from Canvas import Rectangle, Oval, Arc, Window
from colormap import *
import sys
```

```

cmap=SetupColormap0(360)

root=Tk()
cv=Canvas(root, width=401, height=401, borderwidth=0,\
    highlightthickness=0)
ar=Oval(cv, 0, 0, 400, 400)
for i in range(360):
    e=(i+90)%360
    ps = Arc(cv, 0, 0, 400, 400, start=e,extent=1,\
        fill=cmap[i], outline="")

def die(event=0):
    sys.exit(0)

button=Button(cv, text="Quit", foreground="red",\
    background="black", command=die)
Window(cv,380,20,window=button)
cv.pack()

root.mainloop()

```

В строке 15 вычисляется угол начала сектора (как Вы помните, сектор является типом по умолчанию для метода Arc()). По умолчанию для всех секторов, дуг и хорд угол отсчитывается от исходной вертикальной позиции вправо. Чтобы сектор начинался там, где он должен быть, следует выполнить предварительные вычисления, что мы и делаем в строке 15. Ещё одной новой для Вас инструкцией является вызов функции SetupColormap0() в строке 8. Эта функция происходит из модуля colormap, который импортируется в строке 5. Код этого модуля не показан в книге, поскольку для того чтобы выводить цвета в определённом порядке, в модуле происходит преобразование цветовой модели RGB в HLS (hue-lightness-saturation - тон-яркость-насыщенность). Модель HLS значительно удобнее для сортировки цветов, чем модель RGB, но обсуждение этой темы выходит за рамки данной книги. Модуль colormap.py, как и многие другие полезные модули, можно найти на сервере по адресу <http://www.pauahtun.org/>. Чтобы использовать различные методы модуля colormap, совсем не обязательно знать, как они работают. Просто следует указать требуемое число цветов, и любая из функций SetupColormapn() возвратит список заданного числа цветов с дополнительным последним чёрным цветом (зачем добавляется чёрный цвет, Вы поймете, когда мы будем рассматривать наборы Мандельброта в главе 23).

В модели RGB легко построить шкалу серого цвета. В оттенках серого в равной степени представлены компоненты красного, зеленого и синего цветов, отличаясь от других оттенков серого только по яркости. В листинге 21.3 показан код функции `SetupColormap6()`.

Листинг 21.3. SetupColormap6()

```
def SetupColormap6(ncolors):  
    cmap=[]  
    xd=1.0/ncolors  
    for i in range(0,ncolors):  
        r=g=b=0xFF-((i*xd)*0xFF)  
        cmap.append("#%02X%02X%02X"%(r,g,b))  
        cmap.append("#000000")  
    return cmap
```

Как видите, эта функция действительно очень проста. Длина шкалы полутонов определяется как единичная (1.0) и делится на заданное число цветов. В результате получаем значение, приращения яркости при переходе от одного оттенка серого к следующему. Скорее всего, Вы даже не вспомнили об оттенках серого, когда рассматривали переливающийся всеми цветами радуги куб на рис. 21.1. Но если провести диагональ от черного угла к белому, то получится линия, каждая точка которой описывается одинаковыми значениями по осям *R*, *G* и *B*. В строке 6 мы умножаем полученное значение приращения яркости (*xd*) на порядковый номер оттенка в шкале полутонов (1), а затем умножаем полученное значение на 0xFF, чтобы привести числовое значение к формату значений RGB. Полученная шкала оттенков серого показана на рис. 21.3.

***Прим. В. Шипкова:** здесь также пришлось заменить оригинальный рисунок, так как он тоже, мягко говоря ;) устарел.



Рис. 21.3. Шкала серых полутонов

В листинге 21.4 показан код программы создания шкалы серых полутонов.

Листинг 21.4. Программа fountain.py

```
import sys, string

from Tkinter import *
from Canvas import Rectangle, Window
from colormap import *

def die(event=0):
    sys.exit(0)

ncolors=256
w=512
h=64

root=Tk()
cv=Canvas(root, width=w, height=h, borderwidth=0,\
    highlightthickness=0)
cmap=Graymap(ncolors)
wd=w/ncolors
x=y=0
for i in range(ncolors):
    Rectangle(cv, x, y, x+wd, y+h, fill=cmap[i], width=0,\
        outline="")
    x=x+wd
qb=Button(root, text="Quit", command=die)
item=Window(cv, 450, 32, window=qb)
cv.pack()
root.mainloop()
```

Если необходимо, чтобы чёрный цвет в шкале полутонов находился слева, а не справа, достаточно в строке 17 для возвращенной схемы цветов вызвать метод `cmap.reverse()`. Все списки в Python поддерживают набор стандартных методов, таких как `sort()` и `reverse()`. В строке 17 мы также вызываем функцию `Graymap()`, что соответствует вызову уже знакомой Вам функции `SetupColormap6()`. Эта функция возвращает список строковых цветовых значений Tkinter, которые затем используются для установки цвета заливки объектов `Rectangle`, образующих шкалу полутонов. В данном случае цветовые значения соответствуют оттенкам серого, а прямоугольники, представляющие каждый полутон, имеют 2 пикселя в ширину и 64 пикселя в высоту. Подобные шкалы оттенков серого используются во многих приложениях, например в графических редакторах.

Но в наше время полноцветных цифровых изображений построение шкалы цветных полутонов, пожалуй, будет ещё более интересной задачей. Это совсем не сложно, тем более,

что мы уже рассмотрели набор необходимых функций при построении цветного колеса на рис. 21.2. (Если нужно отобразить на экране в цвете одну из сторон куба 'RGB' подумайте, как можно было бы это сделать без методов модуля colormap?) На рис. 21.4 показан результат подмены в строке 17 вызова функции Graymap() на функцию SetupColormap(). (В коде нужно будет сделать ещё некоторые минорные изменения, показанные в листинге 21.5).

***Прим. В. Шипкова: и опять пришлось заменить оригинальный рисунок на более свеженький. ;) Что-то мне подсказывает, придётся описывать цветовые функции отдельно. :)**

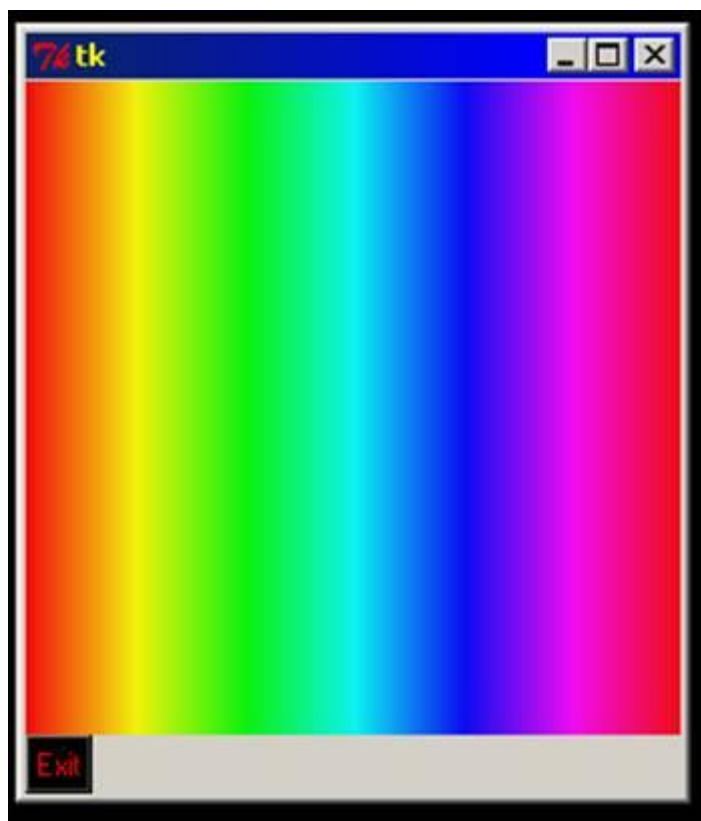


Рис. 21.4. Цветная шкала полутонов

Код программы целиком показан в листинге 21.5.

Листинг 21.5. Программа colorfountain.py

```
import sys, string

from Tkinter import *
from Canvas import Rectangle, Window
from colormap import *

def die(event=0):
    sys.exit(0)
```

```

ncolors=256
w=256
h=256
root=Tk()
frame=Frame(root)
cv=Canvas(frame, width=w, height=h, borderwidth=0,\
    highlightthickness=0)
cmap=SetupColormap0(ncolors)
wd=w/ncolors
x=y=0
for i in range(ncolors):
    Rectangle(cv,x,y,x+wd,y+h,fill=cmap[i],width=0,outline=""
)
    x = x+wd
    qb=Button(frame, text="Exit", command=die,\
        foreground="red", background="black")
frame.pack()
cv.pack(side=TOP)
qb.pack(side=LEFT)
root.mainloop()

```

Данный код почти ничем не отличается от рассмотренного в листинге 21.4. Исключение состоит только в установке ширины и высоты объекта Canvas. Запустите программу `colorfountain.py`, чтобы посмотреть, как выглядит шкала цветных полутонов. В следующих разделах мы воспользуемся ею в качестве фона холста при создании простейшей анимации.

Анимация

В 1972 г. я был студентом колледжа. Помню, как я сидел в "Папа Чарли" (кафе в центре греческих кварталов Чикаго) в компании друзей после утомительных выходных, которые мы провели, роясь в карьерах и железнодорожных насыпях в поисках окаменелостей. Окаменелости нам были нужны для выполнения практического курса по геологии. Напротив нас стояли игральные машины, на мониторах которых видны были летающие филлярдные шары. Я не был любителем этой игры, но не мог устоять перед магией электронной игрушки и сыграл пару игр. Игра мне быстро надоела, но, видимо, с этого момента компьютеры запали в мою душу. До этого единственный компьютер, на котором мне приходилось работать, был монстр, заполнявший собой целый зал и отличавшийся весьма посредственными возможностями. Тут же передо мной стояли сравнительно небольшие машины, которые не только воспроизводили красочные графические изображения, но и позволяли управлять ими. В следующем семестре я проходил курс по зооархеологии. Профессор предложил нам занести в

компьютер базу данных по идентификации нескольких тысяч окаменелых костей животных, которые мы собрали во время экспедиции в Турцию. У профессора были знакомые, которые могли затем выполнить статистическую обработку данных, и он надеялся получить таким образом ошеломляющие результаты. Проект не был доведён до конца, но я вновь столкнулся с компьютерами и с тех пор не расстаюсь с ними.

Видимо, под влиянием юношеских воспоминаний мне пришла в голову идея, что для обучения анимации лучше всего будет написать программу, воспроизводящую движение шаров по бильярдному полю. Предполагалось, что в бильярд должны играть несколько игроков с виртуальными киями. Впрочем, эта идея мне быстро надоела, потому что я не любитель бильярда. Поэтому свою программу я затем преобразовал в поле, по которому вместо шаров летали фигурки древних богов индейцев Майя (надо же мне было применить мою коллекцию тотемных изображений, с которой Вы скоро познакомитесь). А чтобы порадовать глаз, зеленое бильярдное поле мы сделаем переливающимся всеми цветами радуги (Вы научились делать это в предыдущем разделе).

Но, пожалуй, это будет многовато для одной главы. Так что летающих богов оставим до следующей главы, а пока займёмся созданием исходного игрового поля.

Чтобы соответствовать реальности, наш шар при столкновении с краями поля должен отскакивать под тем же углом. Другими словами, угол падения должен быть равен углу отражения. Если шар наткнулся на боковую стенку под углом в 45° , то он должен отскочить под углом в -45° . Вы увидите, что так и происходит (рис. 21.5), когда запустите программу `tkpool.py`, код которой показан в листинге 21.6.

```
import sys, string

from Tkinter import *
from Canvas import Rectangle, Line, ImageItem, \
    CanvasText
from math import *

w=128.0
h=256.0
xp=0
yp=h
xdelta=2.0
ydelta=-2.0
wball=None
filename="white.gif"
```

```

tm=None

img=None
txt=None

def Radians(x):
    return (x*(pi/180.0E0))

def Degrees(x):
    return (x*(180.0/pi))

def die(event=0):
    sys.exit(0)

def moveball(*args):
    global cv, img, h, w, wball, xp, yp, xdelta, ydelta,\
        tm
    cv.move(wball, xdelta, ydelta)
    if xp>=0.0 and yp>=0.0:
        if xp>=w:
            xdelta=xdelta*-1.0
            xp=xp+xdelta
            yp=yp+ydelta
            tm=cv.after(10, moveball)
        else:
            cv.delete(wball)
            tm=None

def chsize(event):
    global w, h, xp, yp, xdelta, ydelta
    w=event.width
    h=event.height

def pause(event=0):
    global tm
    cv.after_cancel(tm)

def shoot(event):
    global cv, img, h, w, wball, xp, yp, tm
    global xdelta, ydelta, filename, txt
    img=PhotoImage(file=filename)
    xp=event.x
    yp=event.y
    xdelta=2.0
    if yp<h/2.0:
        if ydelta<0:
            ydelta=ydelta*-1.0
    else:

```

```

    if ydelta>0:
        ydelta=ydelta*-1.0
    a=yp-h/2.0
    bw=w-xp
# A - угол между направлением полета шара и
# перпендикуляром, опущенным к правому -краю окна
# a - высота получившегося треугольника, сторона a;
# b - длина перпендикуляра, сторона b;
# c - длина гипотенузы, сторона c.
# Все три тригонометрические функции _could_
# возвращают 0
# поэтому их было бы не плохо заключить в
# конструкцию try...except.
A=atan2(a,bw)
deg=Degrees(A)
b=2.0/tan(A)
xdelta=abs(b)
cv.delete(wball)
cv.delete(txt)
wball=ImageItem(cv, xp, yp, image=img)
txt=CanvasText(cv, w-25, h-25, font="Helvetica 14",
text="%02d"%int(deg))
tm=cv.after(10,moveball)

if __name__=="__main__":
    root=Tk()
    cv=Canvas(root, width=w, height=h, borderwidth=0,\
background="#409040", highlightthickness=0)
    cv.bind("<Configure>", chsize)
    cv.bind("<Button-1>", shoot)
    cv.bind("<Button-3>", pause)
    root.bind("<Escape>", die)
    cv.pack(expand=1, fill=BOTH)
    root.mainloop()

```

*Прим. В. Шипкова: и этот рисунок постигла участь прежних
- был заменён на более свежий.



Рис. 21.5. Электронный бильярдный стол

В строках 83–93 создаётся обычное окно Tk с объектом холста внутри. В строке 91 для объекта холста `cv` устанавливаются свойства расширяемости и обязательное заполнение всей поверхности корневого окна-контейнера. В результате пользователь получает возможность изменять размеры окна, перетаскивая мышью его стороны или углы, после чего игровое поле автоматически изменит свой размер в соответствии с новыми размерами окна. Поскольку для работы программы необходимо знать текущие габариты поля, следует определить функцию, которая будет запускаться каждый раз при изменении размера окна. Для этого в строке 92 используется стандартный метод `bind()` для связывания функции `chsize` с событием `"<Configure>"`. В строках 44–47 мы видим, что функция `chsize()` возвращает и записывает для нас текущие размеры ширины и высоты холста.

Метод `bind()` используется также в строке 90 для установки вызова функции `die()` нажатием клавиши `<Esc>`. Таким образом, можно завершить выполнение программы, благодаря чему нам не нужно устанавливать на поле кнопку `Заккрыть` или `Выход`. Обратите внимание, что в этот раз вызывается метод `bind()` корневого окна и именно ему присваивается функция обработки события `die()`. Дело в том, что корневое окно перехватывает все нажатия клавиш и не передает их вложенному объекту холста.

В строке 89 метод `bind()` используется для назначения выполнения функции `shoot()` событию щелчка мыши где-либо в пределах холста. В данном приложении именно на функцию

`shoot()` возложена основная нагрузка. В результате выполнения этой функции из места щелчка мыши будет вылетать шар в направлении середины правого края окна. Ударившись о стенку, этот шар будет отлетать под тем же углом. Ниже мы подробно рассмотрим все нюансы работы функции `shoot()`.

Прежде всего, в строке 52 функция `shoot()` извлекает графическое изображение шара, используя для этого функцию `PhotoImage()`, которую мы рассмотрели в предыдущей главе. В данном месте программе может быть назначен любой графический файл в формате GIF, лишь бы он был в форме квадрата и не очень большой. Следующая задача выполняется в строках 53, 54 и состоит в том, чтобы вернуть координаты щелчка мыши и присвоить их переменным `xr` и `yr`. Эти значения необходимы для выполнения функции `moveball()`. Эта функция вызывает метод `move()` объекта `Canvas`. Методу `move()` в списке параметров передаётся переменная `wball`, которая указывает положение изображения шара на экране. Перемещение шара контролируется двумя другими параметрами `xdelta` и `ydelta`. В физике и математике принято обозначать минимальные векторные изменения (направления или скорости) греческой буквой Δ (дельта), почему мы и выбрали такие названия для переменных. В строках 56-61 определяется направление, в котором должен перемещаться шар. Шар будет перемещаться вверх, если щелчок мышью был сделан в нижней половине поля, или вниз, если щелчок был сделан в верхней половине поля. Поскольку нам нужно также определить угол, под которым будет перемещаться шар, придётся освежить в памяти знания по тригонометрии. Однако обучение Вас основам тригонометрии выходит за рамки этой книги. (Возможно, Вам в этом помогут Web-страницы, приведенные в разделе "Примеры и задания" в конце данной главы.) Сейчас просто поверьте мне, что для выполнения подобных задач Вам будет необходимо знание тригонометрических функций.

Для выполнения тригонометрических функций нужны некоторые исходные данные, такие как длина основания и высота правильного треугольника, описывающего положение объекта. После щелчка мышью мы получаем две координаты x и y , по которым следует построить правильный треугольник. Высота треугольника вычисляется в строке 62: $a = yr - h/2.0$ (в этой программе мы всегда нацеливаем шар в середину правого края окна). Длина основания треугольника вычисляется в строке 63: $bw = w - xr$. На рис. 21.6 показаны традиционные названия сторон и вершин правильного треугольника.

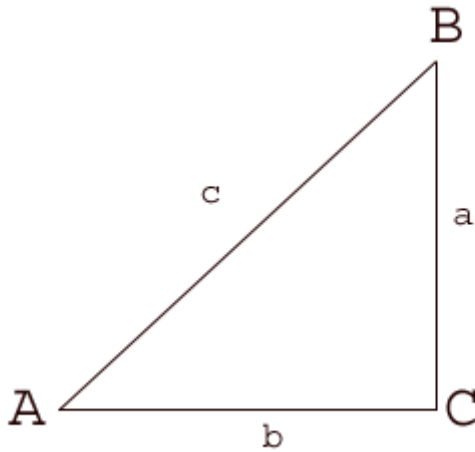


Рис. 21.6. Правильный треугольник

То, что мы называли "основанием треугольника", показано на рисунке стороной b , а высота — это сторона a . Сторона называется гипотенузой (вспомните теорему Пифагора, она Вам как раз сейчас понадобится). Поскольку нам известны длины сторон a и b , не составит труда вычислить угол A , что мы и делаем в строке `71:A=atan2(a,bw)`. Вместо функции `atan2()` можно использовать стандартную функцию `atan()` из модуля `math` (он импортируется в строке `7:from math import *`). Но в этом случае предварительно нужно разделить длину стороны a на b . Функция `atan2()` выполняет это деление за Вас перед тем, как вычислить угол A . Применительно к окну нашего приложения вершина A соответствует точке щелчка мышью, а B — середине правой стороны окна. Вершина C соответствует проекции точки щелчка мышью на правую сторону окна.

Поскольку все функции модуля `math` оперируют и возвращают значения углов, выраженные в радианах, а не в градусах, нам необходима функция преобразования радиан в градусы. Эта функция записана в строках 25, 26. Замкнутая окружность соответствует 360 градусам, или $6,28318530718\dots$ радиан. Это равенство можно выразить формулой $360^\circ = 2\pi$. Для преобразования значений, выраженных в радианах, в углы используется функция `Degrees()`, а для обратного преобразования — `Radians()` (хотя последняя в данном варианте программы не используется). В строке 72 вызывается функция `Degrees()`, и возвращенный результат сохраняется в переменной `deg`.

В строке 74 вычисляется минимальное смещение по оси X с помощью выражения `b=2.0/tan(A)`. Для вычисления используются тангенс определённого ранее угла A и значение минимального

смещения по оси Y , которое устанавливается равным двум пикселям. Смещение должно быть небольшим, чтобы сделать плавным движение шара на экране. Результат этого выражения может быть как положительным, так и отрицательным. Поскольку наш шар всегда должен двигаться вправо, следует использовать абсолютное значение полученного результата, для чего и нужна строка `75:xdelta=abs(b)`. На этом этапе мы выполнили все необходимые предварительные вычисления.

Строки 76, 77 удаляют старое изображение шара и старый текст, если таковые присутствовали на холсте. В строке 78 новое изображение шара устанавливается в холсте по координатам щелчка мышью. В строках 79, 80 создаётся надпись, которая размещается в нижнем левом углу окна и указывает угол, под которым был запущен шар.

За перемещение шара отвечает функция `cv.after(10, moveball)`, которая вызывается в строке 81. Метод `after()` объекта `Canvas` устанавливает таймер 10 миллисекунд, после чего вновь повторяется вызов функции `moveball()`. Возвращаясь к функции `moveball()` в строках 32-42, мы видим, что основная её задача состоит в вызове метода `move()` для объекта `wball`, созданного в строке 78 функцией `shoot()`. Объект смещается по осям X и Y на вычисленные нами ранее значения `xdelta` и `ydelta`.

***Прим. В. Шипкова: если угол будет близок к 90 градусам, будет заметно ускорение шара, а если близок к нулю – будет еле ползти. Дело в том, что программа не контролирует длину гипотенузы. Что, методически, является ошибкой.**

Кроме того, функция определяет, не достиг ли объект правого края окна. Для этого достаточно контролировать только координату x . В строке 36 текущая координата объекта по оси X сравнивается с шириной окна. Если шар достиг края окна, то значение `xdelta` просто умножается на -1 , в результате чего шар начинает двигаться под тем же углом, но влево. Как видите, для программирования зеркального отражения достаточно просто поменять знак смещения по оси X . В результате, если шар подлетел к краю окна под углом 45° , то далее он продолжит путь под углом 135° . Чтобы математически вычислить угол отражения, нужно просто от 180° отнять угол падения.

***Прим. В. Шипкова: поскольку модель шара проста, то простительно, а вообще, нужно учитывать шероховатость стола и бортика стола, точку попадания кия по шару (левее-правее, выше-ниже, полученный вращательный момент). Так что шарик**

Резюме

В этой главе мы изучили основы программирования цвета в модели RGB, а затем научились создавать шкалу полутонов в оттенках серого и в цвете. Мы также научились использовать объект Canvas для создания простейшей анимации и убедились, что для этого нам просто необходимо вспомнить известные со школьной скамьи тригонометрические функции.

В следующей главе мы продолжим работу над анимацией, несколько усложнив нашу задачу, а затем, в главе 23, перейдем к изучению наборов Мандельброта.

Практикум

Вопросы и ответы

Неужели движение бильярдных шаров по полю можно смоделировать с помощью таких простых функций? Тогда можно было бы стать чемпионом, имея в кармане простенький калькулятор.

В действительности вычисление углов — это самая простая часть моделирования игры в бильярд. Кроме того, нам следовало бы учесть вращение шара вокруг своей оси (по вертикали и горизонтали), трение шара о покрытие бильярдного стола, а также дрожание рук во время удара кием по шару. Теория вероятности в этом случае гораздо лучше сможет помочь в выборе победителя, чем простые тригонометрические формулы. Впрочем, если Вы хотите попробовать смоделировать игру в бильярд, я совершенно не хочу Вас останавливать, тем более, если Вы хорошо знакомы с теорией вероятности. Это должно быть очень увлекательно.

В школе тригонометрия казалась мне чем-то непостижимым. Действительно ли тригонометрические вычисления так просто выполнять?

Сегодня выполнение тригонометрических вычислений стало значительно проще, чем это было во времена, когда у людей не было калькуляторов и компьютеров. В школе Вам, вероятно, приходилось пользоваться тригонометрическими таблицами. Действительно, отыскивать каждый раз нужное значение по таблице, например, какому углу соответствует значение тангенса, равное 0,488673541719, очень нудно. Теперь достаточно использовать встроенную в Python функцию `atan()` или `atan2()` (или воспользоваться далеко не самым сложным

калькулятором), чтобы определить искомый угол:
24,4439547804°.

***Прим. В. Шипкова: в моей школе по крайней мере до 1995 года использовали таблицы Брадиса. ;) Да и у меня эти таблицы лежат где-то на полке .**

Контрольные вопросы

1. Какие цвета будут комплементарными для следующих первичных цветов: красного, зеленого и синего?
 - а) Желтый, оранжевый и лиловый.
 - б) Коричневый, розовый и фиолетовый.
 - в) Желтый, пурпурный и голубой.
 - г) Антикрасный, антизеленый и антисиний.

2. Как проще всего изменить на противоположное направление перехода цветового градиента шкалы, показанной на рис. 21.4?
 - а) Нужно изменить угол насыщенности в модуле `colormap`.
 - б) Применить к списку цветовых значений метод `reverse()`.
 - в) Изменить начальный индекс в цикле обращений к списку цветовых значений.
 - г) Выбрать другую цветовую модель.

3. Что произойдет, если изменить знак на противоположный для параметров `xdelta` и `ydelta` функции `moveball()` в программе `tkpool.py`?
 - а) Шар улетит за край окна.
 - б) Шар полетит в сторону левого края окна и отразится от него.
 - в) Шар исчезнет.
 - г) Шар возвратится обратно по тому же пути до точки щелчка мышью, а потом за неё.

Ответы

1. в. Для красного, зеленого и синего цветов (они являются первичными в модели RGB) комплементарными будут желтый, пурпурный и голубой. В других цветовых моделях установлены иные первичные цвета. Например, для модели YMC первичными являются желтый, пурпурный и голубой. Подумайте, как в таком случае в модели YMC будут называться красный, зеленый и синий цвета?

2. б. Поскольку все функции модуля `colormap` возвращают обычные для Python списки, к ним можно применить стандартный метод `reverse()`, чтобы сменить последовательность элементов в списке на противоположную. Вариант с также возможен, но, по-моему, это более сложное решение.
3. г. В результате изменения знаков параметров `xdelta` и `ydelta` на противоположные шар полетит в обратном направлении к точке щелчка мышью.

Примеры и задания

Лучшее место в Internet, с которого можно начать освоение основ программирования цветовой информации, — домашняя страница Charles Poynton's Color Technology, расположенная по адресу <http://www.inforamp.net/~poynton/Poynton-color.html>.

Чтобы освежить в памяти свои знания по тригонометрии, посетите страницу по адресу <http://aleph0.clarku.edu/~djoyce/java/trig/>.

Этот же источник поможет Вам разобраться, почему в тригонометрических функциях проще использовать значения в радианах, а не в градусах.

Попробуйте самостоятельно изменить программу `tkpool.py` таким образом, чтобы переменная `xdelta` всегда была равной 2,0, а `ydelta` — вычислялась.

22-й час

Функции рисования библиотеки Tk

В предыдущей главе мы продолжили изучение функций рисования библиотеки Tk, рассмотрев функции управления цветом и добавление простой анимации. В этой главе мы продолжим заниматься анимацией, а в завершении познакомимся с менеджерами геометрии объектов. Закончив изучение материала этой главы, Вы сможете создавать достаточно сложную анимацию в объекте Canvas; использовать три метода размещения объектов в окне: Place, Pack и Grid.

Ещё анимация

В предыдущей главе я пообещал Вам создать игровое поле, по которому вместо бильярдных шаров будут летать изображения богов индейцев Майя. Давайте начнём с создания

игрового поля. На рис. 22.1 показано, как я его себе представил.

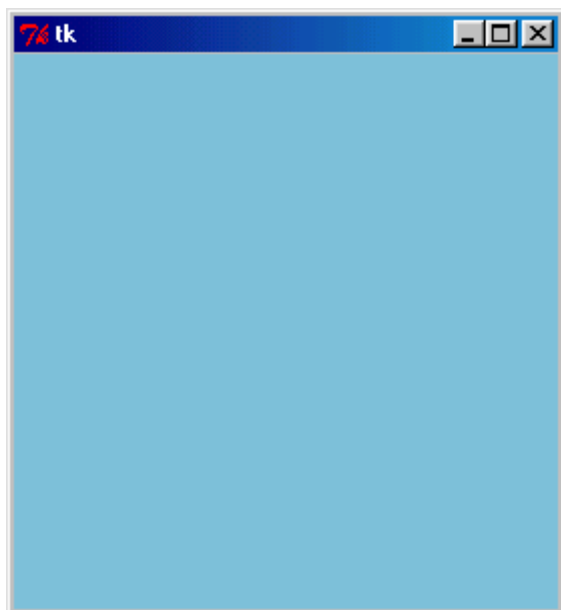


Рис. 22.1. Игровое поле

Выглядит достаточно скучновато, не правда ли? Особенно в монохромном отображении. Запустите на своём компьютере программу `tkfield1.py`, код которой показан в листинге 22.1, чтобы посмотреть, как это поле выглядит на самом деле.

***Прим. В. Шипкова: поле выглядит иначе. Я не стал приводить его здесь, так как это несущественно (поле с вертикальным цветовым градиентом).**

Листинг 22.1. Программа `tkfield1.py`

```
import sys, string, random
from Tkinter import *
from Canvas import ImageItem, Rectangle, CanvasText
from colormap import *

class Glyphs(Frame):
    def die(self,event=0):
        sys.exit(0)

    def __init__(self, parent=None, nobj=0, wwidth=640,\
                 wheight=480):
        self.pause=0
        self.xdelta=[]
        self.ydelta=[]
        self.xmin=16
        self.ymin=16
        self.windowwidth=wwidth
```

```

self.windowheight=wheight
self.xlim=self.windowwidth - self.xmin
self.ylim=self.windowheight - self.ymin
self.xpos=[]
self.ypos=[]
self.img=[]
self.pictures=[]
self.deltav=1
self.parent=parent
Frame.__init__(self, self.parent)
Pack.config(self)

self.nobj=nobj
self.buildglyphs(nobj)

def buildglyphs(self,n=1):
    self.ncolors=128
    self.cmap=SetupColormap1(self.ncolors)

    self.draw = Canvas(self,
width=self.windowwidth, height=self.windowheight)
    self.llabel=CanvasText(self.draw, \
        self.windowwidth-100, self.windowheight-32,\
        text="",font="Helvetica 20")
    self.parent.bind("<Escape>", self.die)
    sr=1+(self.windowheight/self.ncolors)
    x=0
    y=0
    w=self.windowwidth
    for c in self.cmap:
        if c=="#000000":
            break
        item=Rectangle(self.draw, x, y, w, y+sr, fill=c,
            outline="",width=0)
        y=y+sr
    self.draw.pack()

if __name__ == "__main__":
    wwidth=-1
    wheight=-1
    if len(sys.argv)>1:
        nobj=string.atoi(sys.argv[1])
        if len(sys.argv)>3:
            wwidth=string.atoi(sys.argv[2])
            wheight=string.atoi(sys.argv[3])
    else:
        nobj=-1
    root=Tk()
    if wwidth==-1:

```



```

        wwidth=root.winfo_screenwidth()
        wheight=root.winfo_screenheight()
    if nobj==1:
        suff=""
    else:
        suff=" Brothers"
        root.title("Flying Glyph" + suff)
        glyphs=Glyphs(root, nobj, wwidth, wheight)

glyphs.mainloop()

```

Хотя данный код не делает ничего, кроме как выводит на экран радужное поле, в действительности он достаточно функционален. Просто его функциональность нацелена на то, чтобы быть основой для будущей большой программы. Поскольку ожидается, что конечная программа будет достаточно сложной, наш исходный небольшой код мы начинаем с определения класса Glyphs в строке 8. Обратите внимание, что класс Glyphs наследуется от класса графического объекта Frame (рамка), следовательно, и сам является классом графического объекта, предоставляющим доступ к методам класса Frame. Когда класс наследуется от другого класса, то для создания унаследованных переменных-членов в коде дочернего класса нужно явно вызвать метод `__init__()` родительского класса. Это и происходит в строке 28. В строке 29 вызывается метод `pack()`, который необходим для того, чтобы сделать графический объект видимым на экране. В нашем примере этот метод выполнен таким образом, чтобы при реализации класса его экземпляр тут же выводился на экран. В строке 74 создаётся объект класса Glyphs, а в строке 76 для этого объекта вызывается метод `mainloop()`. Как видите, метод `mainloop()` также наследуется пользовательскими графическими объектами от корневого окна, благодаря чему их вывод ничем не отличается от вывода базовых объектов. Это создаёт дополнительную гибкость программирования GUI, в чем мы убедимся, рассмотрев листинг 22.2.

Листинг 22.2. Программа `tkm.py`

```

from Tkinter import *
import sys

def die(event=0):
    sys.exit(0)

x=Button(None, text="Hello, World!", command=die)
x.pack()
x.mainloop()

```


Обратите внимание, что в этом примере кнопка создаётся без предварительного создания корневого окна инструкцией `root=Tk()`, как это было раньше. На экран выводится обычная кнопка, правда, мы не можем вызвать для неё метод `title()` и некоторые другие методы, характерные для окон верхнего уровня. (Как Вы помните, кнопки, создаваемые нами до сих пор, представляли собой окна и владели всей функциональностью окон верхнего уровня.) Тем не менее данная кнопка обладает достаточной функциональностью, чтобы использовать её по прямому назначению в небольших окнах, вроде окон сообщений, или в окнах, поддерживающих ввод данных в текстовых полях.

В строке 42 листинга 22.1 метод `die()` назначается нажатию клавиши `<Esc>`, благодаря чему мы можем обойтись без вывода на экран кнопки "Выход". В строках 67, 68 устанавливается размер дисплея, чтобы по умолчанию вывести окно программы во весь экран. При желании Вы можете открывать окно размером в пол-экрана. Для этого достаточно разделить возвращённые значения высоты и ширины экрана на 2. В строках 47-52 в объект холста выводится множество разноцветных прямоугольников, которые следуют друг за другом, образуя градиентный переход сверху вниз. Подобные градиенты мы уже создавали в прошлой главе. Чтобы настроить градиентные переходы цветов и тонов требуемым образом, следует воспользоваться различными функциями `SetupColormap()` модуля `colormap`. Вспомните, что последним цветом в списке `map` всегда является чёрный. Поэтому завершение цикла инструкцией `break` мы связываем с соответствием текущего значения черному цвету. Обратите также внимание на то, что объект текста, который создаётся в строках 40, 41 для вывода в объекте холста, в действительности будет невидимым на экране, поскольку ему пока что присвоено пустое строковое значение. Этим объектом мы воспользуемся в следующих вариантах программы, впрочем, как и всеми остальными членами класса `Glyphs`.

Первое изменение нашей исходной программы будет состоять в добавлении графических изображений, причём не одного, а сразу двадцати. Кроме того, эти изображения будут выводиться на экран случайным образом. Полный код программы с добавленными графическими изображениями показан в листинге 22.3.

Листинг 22.3. Программа `tkfield2.py`

***Прим. В. Шипкова: текст программы составляет более 120 строк. Соответственно, его здесь нет. Кому надо - [скачать файл](#).**

Первое изменение, которое бросается в глаза, — это пара списков, добавленных в программу в строках 13–26. Это имена графических файлов, которые мы хотим использовать в нашем "Пантеоне летающих богов". Данные файлы следует загрузить с Web-страницы этой книги и скопировать в ту же папку, из которой запускается программа. Все файлы имеют формат GIF. Мой выбор этого формата был обусловлен следующими фактами.

1. Файлы PNG, которые часто рекомендуются как альтернатива файлам GIF, поддерживают прозрачность изображений. В то же время встроенная прозрачность изображений PNG поддерживается далеко не всеми браузерами, используемыми для просмотра Web-страниц. Для поддержания прозрачности в изображениях GIF нужно использовать дополнительные утилиты, такие как Alchemy Mindwork's Gif Construction Set Pro, но в этом случае свойство прозрачности станет переносимым.
2. Метод `PhotoImage()` работает с графическими изображениями только в форматах PPM и GIF, но файлы PPM не поддерживают прозрачность.

Два списка графических файлов отличаются тем, что в списке `smallglyphlist` представлены изображения размером 32x32 пикселя, а в списке `bigglyphlist` — размером 64x64 пикселя. В строке 36 проверяется текущий размер окна программы, и если он меньше 800 пикселей, то выводятся маленькие изображения. В противном случае используются большие изображения. Таким образом, я пытаюсь достичь пропорциональности изображений богов размеру окна программы. Подход, при котором элементы графического интерфейса выбираются из альтернативных наборов и выводятся только после анализа текущего состояния программы, в современном программировании называется *итеративным выводом*, или *пошаговой настройкой*. Программирование итеративного вывода пользовательского интерфейса в других языках выглядит достаточно тяжеловесно, как бы мило и многообещающе не звучали названия специальных средств программирования, зарезервированных для выполнения этих задач. Для выполнения этой задачи предполагается создание множества спецификаций, маркирующих программные блоки. В Python, каким бы большим ни был проект, до создания спецификаций дело, как правило, не доходит. Самый большой проект в Python обычно завершается до возникновения необходимости в спецификации и задолго до Вашего обеда.

Продолжим рассмотрение кода листинга 22.3. После создания объекта графического изображения в строке 77 мы добавляем его в список `img`, откуда затем в любой момент можем его вернуть. А нам они понадобятся позже, когда мы станем

запускать изображения богов в полет по игровому полю. В строках 79-83 программа извлекает первый объект изображения из списка и вычисляет, как близко к краю окна оно может подлететь. По умолчанию координаты объектов на холсте вычисляются по центру соответствующего изображения. Но если мы хотим добиться видимости того, что лики древне-индейских богов подлетают к краю окна и отражаются от него, нужно чтобы центры изображений не подходили к краю ближе, чем на половину самого изображения.

В строке 74 вычисляется значение переменной с именем `deltav`. (Это имя было взято из терминологии ракетостроения, правда, вместо слова *delta* используется греческая буква Δ . Этим термином обозначается ускорение ракеты при старте.) Наши изображения богов будут летать без ускорения, но с разной длиной единичного шага перехода. Маленькие изображения должны перемещаться мелкими шажками, а крупные — делать шаги побольше. Эти значения как раз и заносятся в переменную `deltav`. В строках 84-89 переменным сдвига по осям X и Y присваивается положительное значение `deltav`, если выводимый объект чётный в списке, или отрицательное — если нечётный. В результате одни изображения будут перемещаться на экране слева направо, а другие — наоборот. В строках 91, 92 определяется, где на холсте будет выводиться текущий объект изображения. Метод `random.randint()` возвращает случайные целочисленные значения в указанном диапазоне. В нашей программе этот диапазон вычисляется по текущему размеру окна, с учетом размера используемых графических объектов. Так, если размер окна составляет 256x256 пикселей, случайные значения для координат x и y будут выводиться в диапазоне от 32 до 224. Полученные значения добавляются в списки координат по оси X (`xpos`) и координат по оси Y (`ypos`). И, наконец, в строках 94-96 с помощью метода `ImageItem` в холсте создаётся собственно сам объект графического изображения, который добавляется в ещё один список — `pictures`. После создания списка объектов графических изображений все они будут выведены в холсте вызовом метода `mainloop()` в строке 121. Чтобы запустить эту программу, введите в командной строке окна терминала команду `python tkfield2.py 1256 256`. В результате появится окно программы размером 256x256 пикселей с единственным изображением на нем, как показано на рис. 22.2.

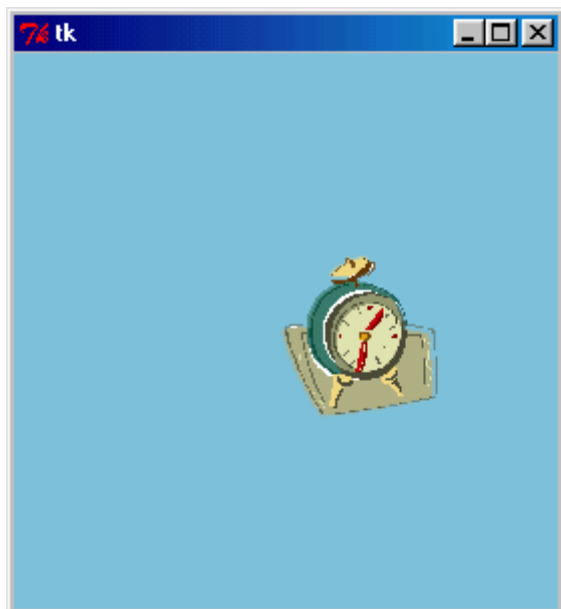


Рис. 22.2. Первая пиктограмма уже на экране, но ещё не летает

При следующем вызове изображение будет выведено в другом месте на холсте. Введите теперь в командной строке `python tkfield2.py 20 256 256` и нажмите `<Enter>`. Результат выполнения программы показан на рис. 22.3.



Рис. 22.3. "Пантеон" заполнен, но боги ещё не летают

Обратите внимание, что когда на экран выводится только одно изображение, то в заголовок окна программы выносится только слово "Пантеон", чтобы у пользователя не возник вопрос, а где же остальные боги. Но если на экран выводится несколько изображений, то в заголовке мы видим: "Пантеон летающих богов". Это ещё один пример итеративного вывода элементов интерфейса, который реализуется в строках программы 114-118. Если запустить программу без аргументов,

то окно приложения откроется во весь экран и будут выведены крупные изображения всех богов.

Теперь мы готовы к добавлению анимации. Представьте себе, что изображения богов, показанные на рис. 22.3, не просто показаны на экране, но беспорядочно движутся, натыкаясь и отскакивая от краев окна. Код программы с анимацией показан в листинге 22.4.

Листинг 22.4. Программа tkfield3.py

***Прим. В. Шипкова: код этой программы больше чем предыдущей – 150 строк. Молча даю [ссылку на файл](#).**

Что нового в этом коде? В строке 99 вызывается метод `moveglyphs()`, определение которого представлено в строках 101-127. В предыдущей главе Вы уже видели пример применения метода `after()`. Особенность его применения состоит в том, что когда функция обработки события вызывается методом `after()`, то в самой функции также должен быть вызов этого метода. Так происходит и у нас. В строке 127 метод `after()` вызывает функцию `moveglyphs()`, а в строке 103 в теле функции `moveglyphs` повторяется тот же вызов метода `after()`. Перед этим в строке 102 проверяется состояние переменной `pause`, с помощью которой можно остановить движение изображений. Как это происходит, мы рассмотрим чуть позже.

Вас может удивить, почему для управления движением всех объектов изображений используется единственный вызов метода `after()`, вместо того, чтобы вызывать его для каждого индивидуального объекта. Проблема состоит в том, что хотя в программах для UNIX можно создавать сколько угодно таймеров, в Windows этот ресурс крайне ограничен. Хотя в Windows 95 само по себе такое ограничение было снято, в действительности мало сделано для того, чтобы эффективно поддерживать эту продекларированную возможность. Эффективность выполнения программы в Windows стремительно снижается после того, как число таймеров станет больше десяти. Но часто в программе для группы объектов удаётся обойтись всего одним таймером, что и продемонстрировано выше в программе.

***Прим. В. Шипкова: на самом деле в документации по Visual Basic написано, что таймеров не может быть более 32. Даже если, речь шла об одном приложении – это хороший показатель того, как не продуман этот вопрос.**

Структура метода `moveglyphs()` достаточно проста и прямолинейна. Этот метод даже проще, чем аналогичная ему

программа tkpool.py, рассмотренная в предыдущей главе. В данном случае мы даже обходимся без импортирования метода math. Дело в том, что в нашей небольшой программе с летающими шарами для вычисления углов использовались тригонометрические функции. Сейчас мы обходимся даже без этих простейших тригонометрических вычислений. Мне было интересно попытаться обойтись без них. Математические вычисления расходуют достаточно много компьютерных ресурсов. Я мог бы поспорить, что в тех игровых аппаратах, которые я видел в 1972 году, наверняка не использовалось никаких тригонометрических вычислений, так как снабжены они были в лучшем случае процессором 6502 CPU и 1 или 2 Кбайт оперативной памяти. Нам, конечно, нет нужды сейчас ограничивать себя такими ресурсами, тем не менее, мне показалось крайне интересным максимально оптимизировать нашу программу.

***Прим. В. Шипкова: 6502 – очевидно речь идёт об одном из первых процессоров фирмы Motorola.**

В предыдущей главе я сообщил, что при ударе движущегося объекта о край окна он отражается по принципу: "угол отражения равен углу падения". Не трудно представить, что если бы наши изображения богов вели себя подобным образом, без изменения направления движения, то очень скоро вошли бы в бесконечно повторяющийся цикл. Понаблюдав за выполнением нашей текущей программы, Вы убедитесь, что такого не происходит. Итак, нет тригонометрических вычислений, нет строгого соблюдения принципа равенства угла падения углу отражения. Что же здесь происходит?

Давайте проанализируем цикл for функции moveglyphs(). Цикл выполняется столько раз, сколько у нас было создано объектов графических изображений. Список объектов изображений формируется в несколько этапов. Сначала в строке 77 создаётся список графических файлов imv, а потом, в строке 94, с помощью метода ImageItem создаются собственно объекты изображений и помещаются в список pictures. При добавлении объектов в список одновременно в строке 96 переменной-члену tags каждого объекта присваивается строковое значение, по которому затем можно будет обратиться к этому объекту в списке. Мы идем самым простым путём: присваиваем переменной tags имя соответствующего графического файла. Все, что мы делаем затем в цикле функции moveglyphs(), – это перемещаем каждый объект с помощью метода холста move() (строка 107) в соответствии с его текущими значениями xdelta и ydelta. При этом для обращения к объектам используются присвоенные им имена.

После очередного смещения объекта мы вычисляем, куда он сместится на следующей итерации (после следующего вызова метода `after()`). Для этого в строке 109 используются текущие координаты объекта и значение его смещения `xdelta`, а результат заносится во временную переменную `tpr`. Если полученное значение остаётся в пределах, заданных переменными `xmin` и `xlim`, то новая координата по оси *X* просто передаётся переменной `xpos` (строка 116). Если же очередная позиция объекта выходит за очерченные рамки, то следует предпринять какие-то действия, предупреждающие выход объекта за пределы окна программы, для чего выполняются строки 111-115. В строке 111 вычисляется угол отражения, точно так же, как мы делали это в предыдущей главе. Затем мы суммируем значение `xdelta` со значением, возвращенным функцией `random.choice()`, и отнимаем 2. (Как Вы, наверное, уже догадались, модуль `random` содержит функции генератора случайных чисел.) Метод `choice()` случайным образом выбирает один из элементов списка, переданного ему с аргументом (в нашем случае — `(0,1)`), и возвращает его. Таким образом, в нашей программе этот метод может вернуть одно из двух значений: 0 или 1. После того как от этого значения отнимается 2, получаем -1 или -2, что и прибавляем к текущему значению `xdelta`. Таким образом, после каждого удара объекта о стенку мы немного изменяем значение `xdelta`, что равносильно изменению стороны *b* нашего мнимого треугольника, который мы рассматривали в предыдущей главе на рис. 21.6. Чтобы напомнить Вам, что это за треугольник, давайте внимательно рассмотрим движение объектов по экрану. Обратите внимание, что они никогда не движутся точно по вертикали или по горизонтали, а всегда под углом. Можно сказать, что они движутся по гипотенузе правильного треугольника, катетами которого являются `xdelta` (сторона *b*) и `ydelta` (сторона *a*). Наши значения приращений `xdelta` и `ydelta` — это всегда небольшие целые числа, поэтому рассматриваемый нами треугольник также достаточно маленький. Поскольку значения `xdelta` и `ydelta` изменяются только во время ударений объекта о край окна, в пределах окна объекты движутся прямолинейно. Но отражения от стенок теперь уже не соответствуют принципу "угол отражения равен углу падения". Иногда Вы увидите, что объект отражается под более острым углом, иногда под более тупым и движется медленнее. Первый случай происходит, когда, например, `xdelta` принимает значение 1, а `ydelta` — 9, тогда как второй случай может соответствовать значениям 2 для `xdelta` и 1 для `ydelta`.

В строках 119-124 повторяется та же операция, только для переменной `ydelta`, или, если хотите, для стороны *a* мнимого треугольника. Теперь Вы, наверное, уже поняли, как нам

удастся обходиться без тригонометрических функций и почему объекты не движутся по замкнутому циклу.

***Прим. В. Шипкова: приведённый пример, имхо, является классикой того случая, когда можно обойтись без синусов и косинусов. :)**

Добавим теперь ряд последних изменений в код программы, как показано в листинге 22.5.

Листинг 22.5. Программа tkfield4.py

***Прим. В. Шипкова: код этой программы больше чем предыдущей – 170 строк. Молча даю [ссылку на файл](#).**

Новыми в этом коде являются два метода – `snap()` и `lifter()` (строки 12-26). В строках 79, 80 эти методы назначены для обработки событий щелчков кнопками мыши. Метод `snap()` назначен событию "<Button-1>" и служит для остановки движения всех объектов. Этот метод устанавливает значение переменной `pause`, которая проверяется в функции `moveglyphs()` перед выполнением новой итерации. Если значение `pause` равно 1, то функция вызывает метод `after()`, после чего выполняется инструкция `return`. Поэтому, несмотря на предельную простоту метода `snap()`, его влияние на выполнение программы огромно.

Второй новый метод `lifter()` выполняет две задачи. Он назначен для обработки события "<Button-3>", что соответствует щелчку правой кнопкой мыши. После щелчка мышью в пределах окна программы строка 19 проверяет, находится ли в данный момент под указателем мыши какой-либо объект изображения. Если это так, в переменной `x` окажется больше одного элемента (условие, проверяемое в строке 20), причём первый элемент набора будет содержать имя файла текущего объекта, которое мы назначили переменной-члену `tags` при создании объекта изображения с помощью метода `ImageItem`. В этом случае данная строка текста выводится на холст методом `CanvasText` в строке 76. Поскольку поймать движущийся объект и щелкнуть на нем правой кнопкой мыши может оказаться сложно, щелкните предварительно левой кнопкой мыши, чтобы методом `snap()` "заморозить" снующие объекты, после чего щелкните на понравившемся объекте правой кнопкой мыши. Результат показан на рис. 22.4.



Рис. 22.4. Возвращение имени файла объекта изображения

Нашу программу можно дорабатывать и дальше, снабжая её все новыми и новыми возможностями. Некоторые задачи, над которыми Вы можете поработать самостоятельно, предложены в разделе "Примеры и задания" в конце главы.

В следующих разделах мы рассмотрим использование специальных методов размещения объектов в окне.

Менеджеры геометрии объектов

Библиотека Tkinter содержит три метода, с помощью которых можно управлять размещением объектов внутри других объектов-контейнеров, изменять размеры встроенных объектов и позиционировать их относительно друг друга. Я не собираюсь тратить на эту тему много времени, во-первых, потому, что с одним из методов Вы уже познакомились при изучении графических объектов библиотеки Tkinter, а во-вторых, поскольку их использование предельно просто и не требует создания сложных конструкций.

Ниже перечислены три метода управления размещением, называемых ещё менеджерами геометрии объектов, в порядке их усложнения:

- `place`;
- `pack`;
- `grid`.

Поскольку Python является объектно-ориентированным языком, в котором все графические объекты являются классами, унаследованными от общего базового класса, данные методы наследуются всеми классами графических объектов. В

других языках программирования такие методы создаются отдельно с использованием средств программирования графического интерфейса и требуют передачи им в параметрах указателей на графические объекты, над которыми следует выполнять операции. В Python эти методы можно вызвать в любой момент для любого графического объекта как обычный метод `place()`, `pack()` или `grid()`.

Метод `place()`

Несмотря на то что метод `place()` имеет простейший интерфейс, его использование может вызвать некоторые затруднения. С его помощью выполняют следующие задачи:

- размещают объекты по абсолютным координатам *x* и *y*;
- отслеживают координаты объекта относительно родительского окна или других объектов.

Как правило, в программах не устанавливаются жёсткие координаты объектов в окне, иначе у пользователя не будет никакой возможности изменить их во время выполнения программы. Для изменения координат объектов используется метод `place`, который вызывается непосредственно для изменяемого объекта. Например:

```
x = Button(root)
text="Кнопка"
x.place(x=10,y=10)
```

В результате выполнения этого кода кнопка будет помещена на расстоянии 10 пикселей от верхнего и левого краев объекта-контейнера. Не менее полезна возможность управления положением объекта в относительных координатах. В листинге 22.6 показано определение класса, в котором метод `place()` используется для того, чтобы разместить вложенные графические объекты по периметру окна-контейнера.

Листинг 22.6. Относительное размещение объектов в окне

```
class Placer:
    def __init__(self, w):
        global death
        opts=[(N,0.5,0), (NE,1,0), (E,1,0.5), {SE,1,1}, \
              (5,0.5,1), (SW,0,1), (W,0,0.5), (NW,0,0), \
              (CENTER,0.5,0.5)]

        for an, xx, yy in opts:
            item=Button(w, text=an)
            item.place (relx=xx, rely=yy, anchor=an)
```

```

if an == CENTER:
    item["command"] = die
    item["image"] = death
else:
    item["command"] = lambda x=an
prn(x)

```

По нумерации строк Вы видите, что в листинге 22.6 представлена часть большой программы tkgeom.py. На рис. 22.5 показано окно, создаваемое классом Placer.

***Прим. В. Шипкова: уже традиционно для графики рисунок пришлось заменить на более актуальный.**



Рис. 22.5. Относительное размещение графических объектов внутри окна с помощью метода place

При относительном размещении объектов в окне длина сторон окна принимается равной 1 (не сантиметру и не метру, а абстрактной единице). Координаты объекта в окне в этом случае определяются не в пикселях, а в десятых долях от длины стороны окна. Список `opts` содержит элементы, представленные наборами из трёх аргументов, необходимых методу `place()`. Этот метод вызывается в строке 76 в теле цикла `for`. При первой итерации цикла в теле цикла происходит распаковка первого элемента списка `opts` и значения набора присваиваются соответственно переменным `an`, `xx` и `yy`. Эти переменные передаются методу `place` в качестве аргументов. При этом используется следующий синтаксис

(имена переменных заменены соответствующими значениями первого набора списка `opts`):

```
item.place(relx=0.5,rely=0,anchor=N)
```

Эта запись означает: отцентровать объект по верхнему (северному — `n`) краю окна. Как Вы видите на рис. 22.5, кнопка `n` действительно находится посередине верхнего края окна. В строке 77 проверяется значение параметра `anchor` текущего объекта, и если это значение `CENTER`, то событию щелчка на кнопке назначается выполнение функции `die()`, а вместо текста на кнопке выводится рисунок из файла, объект которого присваивается параметру `image`. В результате в центре окна вместо тоскливой кнопки "Выход" мы получили яркий значок, на котором изображен бог смерти индейцев Майя. Возможно, интерфейс получился не очень дружелюбным, так как назначение кнопки не столь очевидно, как если бы на ней просто было бы написано "Выход", но моя цель в данном случае состояла в том, чтобы показать Вам способ подмены текста в объекте кнопки и ярлыка на пиктограмму, хранящуюся в графическом файле. Причем если объекту уже присвоена строка текста, то она автоматически будет замещена изображением сразу же после инициализации свойства `image`. Но только предварительно Вам нужно создать объект изображения с помощью метода `PhotoImage()`. (В листинге 22.6 не показана строка с методом `PhotoImage()` по той причине, что эта операция в программе `tkgeom.py` выполнялась в начале кода.) Проанализируйте другие элементы списка `opts` и постарайтесь представить, где метод `place()` разместит в окне соответствующие кнопки и как они будут выглядеть. Обратите внимание, что если изменить размеры окна, относительное размещение кнопок в окне сохранится прежним.

Метод `pack()`

Как Вы видели во всех предыдущих примерах, метод `pack()` вызывается каждый раз перед выводом графического объекта на экран. Если метод `place()` удобно использовать для размещения объектов в окнах в абсолютных или относительных координатах, то метод `pack()` особенно полезен в тех случаях, когда нужно создать ряды и колонки объектов, или, другими словами, создавать "упаковки" объектов (отсюда и название `pack`), определённым образом ориентированные в корневом окне. Код класса `Packer` все той же программы `tkgeom.py` показан в листинге 22.7.

Листинг 22.7. Класс `Packer`

```
class Packer:
```

```

def __init__(self,w):
    global death
    n=0
    self.a=Frame(w)
    self.a.pack(side=TOP,expand=1,fill=BOTH)
    for i in range(3):
        item=Button(self.a, text='n')
        item.pack(side=LEFT, expand=1, fill=BOTH)
        item["command"] = lambda x=n : prn(x)
        n=n+1
    self.b=Frame(w)
    self.b.pack(side=TOP, expand=1, fill=BOTH)
    for i in range(3):
        item=Button(self.b, text='n')
        item.pack(side=LEFT, expand=1, fill=BOTH)
        if i==1:
            item["command"]=die
            item["image"] = death
        else:
            item["command"]=lambda x=n : prn(x)
        n=n+1
    self.c=Frame(w)
    self.c.pack(side=TOP, expand=1, fill=BOTH)
    for i in range(3):
        item=Button(self.c, text='n')
        item.pack(side=LEFT, expand=1, fill=BOTH)
        item["command"]=lambda x=n : prn(x)
        n=n+1

```

В программе создаются 9 кнопок, как и в классе Place, но в данном случае создаются три рамки (self.a, self.b и self.c), в каждую из которых добавляются по три кнопки. Центральной кнопке второй рамки назначаются выполнение функции die() и объект изображения. В итоге мы получаем стек из трёх рамок, каждая из которых представляет собой стек из трёх кнопок. Окно, полученное в результате реализации класса Place, показано на рис. 22.6.

***Прим. В. Шипкова: рисунок заменён.**



Рис. 22.6. Ряды объектов, созданные с помощью метода `pack()`

Кнопки 0, 1 и 2 принадлежат рамке `self.a`, кнопка 3, кнопка с пиктограммой и кнопка 5 принадлежат рамке `self.b`, а кнопки 6, 7 и 8 – рамке `self.c`. Размеры и форму окна можно произвольно изменять с помощью мыши. Посмотрите, как себя при этом будут вести вложенные кнопки. Они меняют свой размер в тех же пропорциях, по-прежнему заполняя всю поверхность корневого окна. Это происходит потому, что для всех объектов рамок и объектов кнопок были установлены следующие свойства: `expand=1` и `fill=BOTH`. Попробуйте изменить значение свойства `fill` для всех объектов и посмотрите, как изменится поведение окна. Например, попробуйте присвоить этому свойству значения `None`, `X` и `Y`.

Метод `grid()`

Это, пожалуй, самый интересный и полезный из всех методов управления размещением объектов. Этот метод использует функции рисования таблицы из надстройки `troff`. Если эти названия Вам ничего не говорят, не волнуйтесь. Пока что детали работы метода Вам не нужны, сконцентрируйтесь на принципах его использования. Метод `grid()` разбивает окно на конструкцию из строк и столбцов. Часто даже не нужно вычислять самостоятельно, сколько строк и столбцов нужно. Можно написать код, который будет автоматически вычислять значения аргументов метода `grid()`. Чтобы разобраться в принципах использования этого метода, рассмотрим листинг 22.8, в котором показано определение класса `Gridder` в программе `tkgeom.py`.

Листинг 22.8. Класс `Gridder`

```
class Gridder:
```

```

def __init__(self ,w):
    global death
    names=[ "A","B","C", "D" ,"E" ,"F",
            "G","H","I",]
    n=0
    for i in range(3):
        for j in range(3):
            item=Button(w,text=names[n] )
            if names [n]=="E":
                item["command"]=die
                item["image"]=death
            else:
                item["command"]=lambda x=names[n] : prn(x)
            item.grid(row=i , column= j , sticky=E+W+N+S )
            n = n+1

```

Мы вновь создаём 9 кнопок с помощью двух вложенных циклов `for`. Обратите внимание на вызов метода `grid()` в строке 28, в результате которого каждой кнопке задаётся соответствующая ячейка в таблице. В этой же строке происходит инициализация свойства `sticky` набором константных значений, указывающих, что форма объектов может произвольно изменяться в пределах ячейки во всех направлениях (константы `E+W+N+S` означают на восток, запад, север и юг). На рис. 22.7 показано окно, созданное классом `Gridder`.

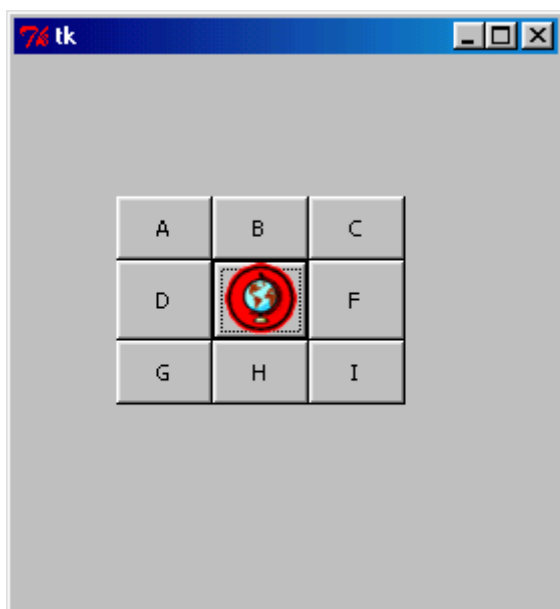


Рис. 22.7. Таблица кнопок, созданная с помощью метода `grid()`

Вам, наверное, показалось, что результат выполнения метода `grid()` не сильно отличается от результата выполнения

метода `pack()` (см. рис. 22.6). Но в действительности отличия существенные. Мы убедимся в этом, когда создадим сложную конструкцию объектов в корневом окне. Например, попробуем создать ещё одну строку таблицы с помощью метода `grid()` и вместо одного объекта добавим в эту строку пакет объектов, созданный с помощью метода `pack()`. Соответствующий код показан в листинге 22.9.

Листинг 22.9. Совместное использование методов `pack` и `grid`

```
class Gridder:
    def __init__(self, w):
        global death
        names=["A","B","C", "D","E","F", "G","H","I",]
        n=0
        for i in range(3):
            for j in range(3):
                item=Button(w,text=names[n])
                if names[n]=="E":
                    item["command"]=die
                    item["image"]=death
                else:
                    item["command"]=lambda x=names[n] : prn(x)
                    item.grid(row=i, column=j, sticky=E+W+N+S)
                n=n+1
        f=Frame(w)
        b1=Button(f, text="Левая кнопка")
        b1["command"]=lambda x="left" : prn(x)
        b2=Button(f,text="Правая кнопка")
        b2["command"] = lambda x="right" : prn(x)
        b1.pack(side=LEFT)
        b2.pack(side=RIGHT)
        f.grid(row=3,column=0,columnspan=3)
```

Результат выполнения этой программы показан на рис. 22.8.

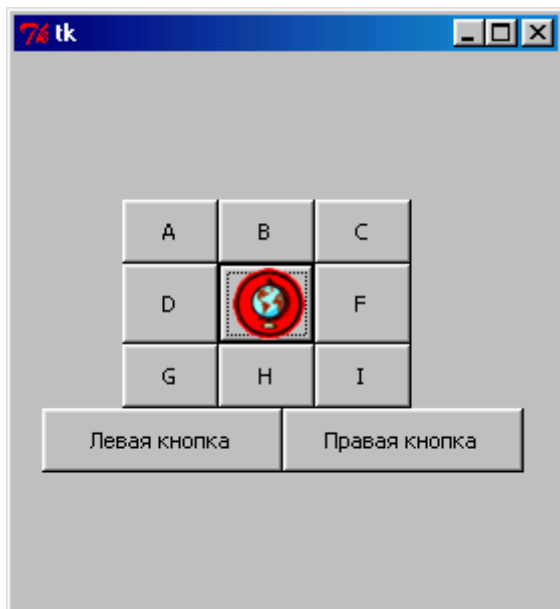


Рис. 22.8. Совместное использование методов `grid()` и `pack()`

Было бы сложно также выровнять наборы из трёх и двух кнопок только с помощью одного метода `pack()`. При использовании данного подхода будьте внимательны во время определения объектов-контейнеров. Так, кнопки от А до I и рамка `f` принадлежат коренному окну `w`, тогда как две нижние кнопки принадлежат рамке `f`. Если в параметрах объектов кнопок `b1` или `b2` вместо объекта-контейнера `f` указать `w`, то между окном и рамкой возникнет неразрешимый конфликт. Один метод управления размещением объектов можно использовать внутри другого, но их нельзя использовать в программе таким образом, чтобы они ссылались на один и тот же объект-контейнер.

В следующей главе при создании программы `mandelbrot.py` мы вновь воспользуемся конструкцией из метода `pack()`, вложенного в метод `grid()`.

Резюме

В этой главе нам удалось создать программу с достаточно сложной анимацией, и Вы познакомились с тремя менеджерами геометрии объектов: `pack()`, `place()` и `grid()`. В следующей главе мы займёмся разработкой достаточно сложной программы, выполняющей вычисления наборов Мандельброта и построением на их основе сложных графических изображений.

Практикум

Вопросы и ответы

Является ли Alchemy Workshop единственной компанией, создающей утилиты для поддержания прозрачности в изображениях формата GIF?

Нет. Вопросами поддержания прозрачности в графических изображениях и анимацией занимается также компания CompuPic Photodex. Причем её утилиты поддерживают работу с файлами разных форматов, а не только GIF. Правда, среди этих утилит нет таких, которые распространялись бы бесплатно. В Internet можно найти много подобных утилит, но вопрос их легальности остаётся открытым, поэтому я не хотел бы давать Вам каких-либо рекомендаций, кроме двух вышеупомянутых компаний.

Код листинга 22.9 довольно сложен. Нельзя ли упростить его каким-либо способом и обойтись без метода `pack`?

Если первая строка с кнопками А, В и С была разделена на 3 колонки, уже невозможно сообщить методу `grid()` о том, что последнюю строку следует разделить на 2 колонки. Если Вы предпочитаете выполнить эту задачу только с помощью метода `grid()`, то в нём следует установить деление строки на 6 колонок, после чего для кнопок первых трёх строк объединять по 2 ячейки с помощью метода `columnspan`, а в последней строке с помощью этого же метода объединить по 3 ячейки вместе. Но в результате код получится ещё сложнее, чем показанный в листинге 22.9.

Контрольные вопросы

1. Чем были заменены тригонометрические функции в программе `tkfield4.py`?
 - а) Генератором случайных чисел.
 - б) Предустановленными координатами x и y .
 - в) Логарифмической таблицей.
 - г) Альтернативными математическими вычислениями.
2. Почему в программе `tkfield4.py` мы не создавали отдельные таймеры для всех перемещающихся объектов графических изображений?
 - а) Нам бы пришлось 20 раз переписать один и тот же код.
 - б) Мы так и поступили, просто в программе это выглядит как один таймер.
 - в) В Windows возможность одновременного использования программе нескольких таймеров сильно ограничена.
 - г) Мы сделали так для наглядности кода, хотя в реальной программе следовало бы снабдить таймером каждый объект.

3. Каким образом в программе tkfield4.py останавливается движение пиктограмм по экрану?
- а) Удаляется вызов таймера.
 - б) Переменным xdelta и ydelta присваивается одно и то же значение.
 - в) Устанавливается флаг, который сообщает функции обработки ситуации, что необходимо завершить работу без выполнения каких-либо действий.
 - г) Устанавливается флаг, который сообщает функции обработки ситуации о переустановке себя как функции обработки ситуации и завершении работы до выполнения функций перемещения объектов.

Ответы

1. а. Умелое использование генератора случайных целых чисел из модуля random позволило нам обойтись без вычислений тригонометрических функций.
2. в. Несмотря на то что начиная с Windows 95 официально было объявлено о снятии ограничений на число таймеров, используемых одной программой, в действительности одновременное использование более десяти таймеров существенно снижает производительность компьютера. Поэтому при программировании для Windows число таймеров в программе следует сократить до минимума.
3. г. Щелчок левой кнопкой мыши где-либо в пределах окна программы устанавливает флаг, который распознается функцией обработки события. В результате эта функция восстанавливает себя как функцию обработки события и прекращает свою работу. Новый щелчок мышью снимает флаг.

Примеры и задания

Чтобы узнать больше о пантеоне богов Майя и познакомиться с их изображениями, посетите узел по адресу <http://vww.pauahtun.org/days.html>. Здесь Вы найдёте изображения духов 20-ти дней священного календаря Майя.

Домашняя страница Alchemy Mindworks находится по адресу <http://www.mindworkshop.com/alchemy/alchemy.html>. Лицензионный договор на использование утилит этой компании для преобразования графических файлов GIF можно найти по адресу <http://www.mindworkshop.com/alchemy/lzw.html>. С помощью программы Gif Construction Set Pro также можно создавать анимацию на основе файлов GIF. Домашняя страница компании CompuPic Photodex находится по адресу <http://www.photodex.com>.

Попробуйте сделать так, чтобы окно программы tkfield4.py можно было произвольно изменять во время выполнения. Подсказка: не забудьте, что помимо объектов графических изображений следует учесть все объекты прямоугольников, из которых создаётся игровое поле.

Попробуйте вместо изображений богов Майя создать и добавить свои собственные графические файлы.

23-й час

Набор Мандельброта

В предыдущей главе мы завершили рассмотрение функций рисования и менеджеров геометрии объектов библиотеки Tk. В этой главе мы применим на практике знания, полученные в предыдущих шести главах, и создадим завершённое приложение для расчёта и вывода наборов Мандельброта (Mandelbrot sets), а также связанных с ними наборов Джулия (Julia sets).

Набор Мандельброта.

Это было в 1985 г., только вышел августовский номер *Scientific American*. На следующее утро большинство инженеров компании, в которой я работал, принесли с собой написанные на скорую руку простенькие программы Мандельброта, а к концу недели у каждого уже была своя такая программа. В следующий понедельник тем, кто пытался выполнить какую-либо полезную работу на любом из семи компьютеров компании, оставалось только чертыхаться, поскольку компьютеры зависали из-за того, что на них несколько программ одновременно вычисляли наборы Мандельброта.

Я думаю, что та же самая картина повторилась во всех компаниях страны, где работали программисты, хотя в нашей компании концентрация инженеров-компьютерщиков на единицу простого населения была, пожалуй, самой высокой. В то время наиболее мощной машиной с графическим интерфейсом, к которой я имел доступ, была Intelecolor, способная выгружать информацию со скоростью 9 600 бод (1 бод = 1 бит/с). И даже на этом чуде техники вывод набора Менделброта занимал не менее 5 минут. А такому давно вымершему динозавру, как Gould NP1, требовалось несколько часов для выполнения той же непростой работы. Компьютер буквально дымился, и это при том, что палитра состояла только из восьми выводимых цветов. Я пытался уговорить шефа подключить сетевую карту на мой терминал, но он сказал: "Обойдешься, я знаю, зачем тебе это нужно!" И он был прав.

Я полагаю, что в течение нескольких месяцев после выхода этого номера *Scientific American* 90–95% национальных компьютерных ресурсов было потрачено на выведение наборов Мандельброта. Изучая материал данной главы, Вы сможете испытать то же удовольствие, которое испытывали мы.

Если вычисление наборов Мендельброта реализовать на языке C, то программа будет выполняться относительно быстро. Конечно, следует учесть, что в 1985 г. самый мощный Gould NP1 имел быстродействие, как мы недавно подсчитали с друзьями, аж 33 МГц. И это был один из мощнейших четырёхпроцессорных компьютеров. Поэтому программы вычисления наборов Мандельброта можно было писать только на C, иначе программы выполнялись бы сутками. Но в наше время, когда быстродействие большинства домашних персональных компьютеров перевалило за 400 МГц, мы можем позволить себе реализовать такую программу полностью на языке Python. Безусловно, что в таком виде программа будет выполняться медленнее, чем если бы она была написана на C. Для компьютеров большинства читателей время выполнения нашей программы, показанной далее в этой главе, составит где-то около 5 минут. Разобравшись в принципах решения задачи в Python, Вы сможете разыскать в Internet более быстрый вариант программы. Для создания программ, подобных той, что приведена в качестве примера в этой главе, могут оказаться полезными дополнительный модуль *NumPy*, который можно загрузить с сервера <http://numpy.sourceforge.net/>, и модуль вывода окон в Windows *wxPython* (<http://alldunn.com/wxPython/>). Рассмотрение этих модулей выходит за рамки данной книги.

Я уверен, что Вы уже видели в своей жизни графику, построенную на основе наборов Мандельброта. Если нет, то один пример показан на рис. 23.1.

Это изображение было создано лично Беноитом Мандельбротом (Benoit Mandelbrot), описавшим в 1980 г. математический подход вычисления наборов его имени, и подарено им фантасту Артуру С. Кларку (Arthir C. Clarke). Др. Кларк воскликнул, что наборы Мандельброта вмещают в себе больше, чем Вселенная, или, по крайней мере, являются её отражением. Этой идеей он поделился со Стефаном Хавкином (Stephen Hawking). В поддержку др. Хавкин привел пример минимальной длины Планка. (*Planck Length* – 1.616×10^{-33} см, меньше которой не может быть ни одна элементарная частица. Меньше кварка – это одна из фундаментальных констант Вселенной.) Тем не менее в математической модели Мандельброта могут существовать частицы меньшего размера, по крайней мере виртуально. Единственный, но безусловный предел – это

минимально допустимое значение числа с плавающей запятой на Вашем компьютере.

***Прим. Венедикта Ли:** Здесь, после сверки с оригиналом, заменены рис. 23.1, 23.3 и добавлены отсутствовавшие фрагменты текста и рисунки 23.4-23.9.

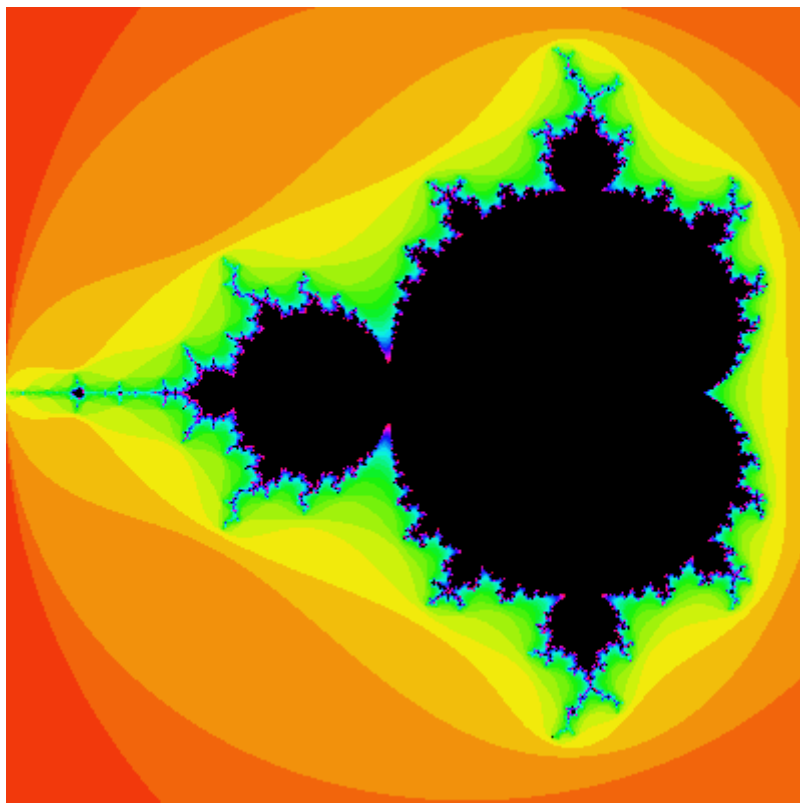


Рис. 23.1. Графическое отображение набора Мандельброта

Математическая сторона вопроса

Формула вычисления набора Мандельброта чрезвычайно проста. В общем виде она выглядит так:

$$z = z^2 + c$$

Некоторые соглашения остались невидимыми в уравнении. Значения z и c — это комплексные числа. Важным моментом также является то, что данное уравнение применяется для вычисления цвета каждого пикселя изображения. Для наборов Мандельброта принято, что изображение формируется в системе координат размером $-2,0 \times 2,0$ по осям X и Y с точкой $(-2,0, -2,0)$ в верхнем левом углу. Напомним, что комплексные числа состоят из двух компонент — реального и мнимого. Графически принято отображать комплексное число как точку в системе координат, где по оси X откладывается реальная составляющая, а по оси Y — мнимая. Любая точка в этом пространстве — комплексное число, такое как c . В Python

создать комплексное число очень просто: $s = \text{complex}(x, y)$. Для наглядности система координат комплексных чисел показана на рис. 23.2.

Как в любой другой системе координат, оси X и Y в системе комплексных чисел бесконечны. Но нам для графического отображения набора Мандельброта нет необходимости вычислять все множество комплексных чисел, а достаточно ограничиться только набором точек, выводимых на экран. В программе, представленной ниже в этой главе, для показа графики я выбрал холст размером 400х400 пикселей, т.е. в нашей системе координат по осям X и Y отложено по 400 точек. Число точек на видимой части оси координат называется *разрешением*. Таким образом, в нашей системе разрешение по оси X и по оси Y равняется 400, а в целом на экран выводится 160000 точек.

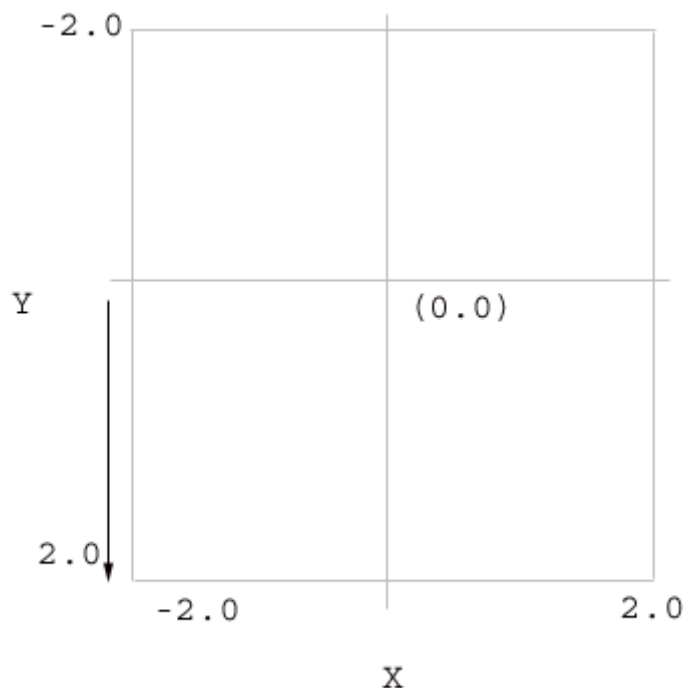


Рис. 23.2. Система комплексных чисел

Увеличивать разрешение без видимой нужды не рекомендую, ведь уравнение $z = z^2 + c$ многократно выполняется для всех выводимых пикселей. Но мы до сих пор не знаем, что такое z . Это значение можно представить как третье измерение – расстояние от точки до плоскости, в которой лежит набор Мандельброта. Первое, что мы делаем, – это определяем для каждого пикселя, принадлежит ли он набору. Если да, то пиксель выводится чёрным цветом. Если нет, то цвет пикселя определяется по тому, насколько он отдалён от плоскости набора Мандельброта. Но как определить, принадлежит ли

пиксель (или точка) этому набору? Для каждой точки мы решаем указанную выше формулу и следим, как увеличивается расстояние от плоскости $(0, 0j)$. Затем применяем следующий алгоритм.

1. Точки, которые никогда далеко не удаляются от плоскости набора, принадлежат этому набору и окрашиваются в чёрный цвет.
2. Точки, которые стабильно удаляются от плоскости набора в бесконечность, не принадлежат набору, а их цвет определяется скоростью удаления.

Как Вы знаете, это недостойное занятие, издеваться над компьютером, оперируя такими понятиями, как "никогда" и "бесконечность". Поэтому мы задаём некоторые произвольные пределы, в которых определяется цвет пикселей. Первый "верстовой столб" на пути от плоскости набора установим по длине выводимой части оси координат. Так как значения на оси X изменяются в пределах от $-2,0$ до $2,0$, то первое граничное значение z можно сделать равным $4,0$. Но как же не хочется иметь дело с отрицательными значениями. Проще будет учитывать абсолютное расстояние от точки до плоскости, которое в этом случае будет изменяться от 0 до $2,0$. Для этого после вычисления расстояния по формуле $z=z^2+c$ используется функция `abs()`. Обратите внимание, что функция `abs()` в качестве параметра принимает числовые значения любого типа, независимо от того, был импортирован модуль `math` или `cmath`.

Для обычных чисел функция `abs()` просто удаляет значение знака. Так, `abs(-4)` возвращает 4 , а `abs(3.14159)` — 3.14159 . Вычисление абсолютного значения комплексного числа выполняется сложнее. Сначала мнимая и реальная составляющие возводятся в квадрат, затем суммируются, а от результата берется корень. Например, при вычислении абсолютного значения комплексного числа $(-0.5, 1j)$ выполняются следующие действия:

```
sqrt((-0.5*-0.5)+(1*1))
sqrt((0.25)+(1))
sqrt(1.25)=1.11803398875
```

Каждую из показанных выше строк можно выполнить в окне интерпретатора по отдельности, в любом случае результат будет один и тот же. Только помните, что в данном случае мы имеем дело не с комплексным числом, а с его имитацией. А теперь попробуем вычислить абсолютное значение действительно комплексного числа. Введите следующие строки в окне интерпретатора:


```
c=0.5+1j
abs(c)
```

В результате получим то же самое число: 1.11803398875 (Если у Вас получилось что-то другое, то либо Вы работаете с весьма древней операционной системой с устаревшим способом представления чисел с плавающей запятой, либо установка Python на Вашем компьютере прошла катастрофически неудачно.) В языке C, где нет встроенных комплексных чисел, для вычисления абсолютного значения комплексного числа потребуется самостоятельно написать код, что усложнит всю программу. К счастью, в Python приведённое выше выражение можно использовать для возвращения искомого значения таким, как оно есть, поэтому я избавлен от необходимости рассказывать Вам о том, как много проблем и побочных явлений ожидает того, кто пытается работать с комплексными числами в других языках программирования. (Раз уж я взялся за написание этой книги, то чувствую своим долгом убедить Вас, что Python – самый лучший язык программирования!)

***Прим. В. Шипкова: и я, и я, и я того же мнения. (с)
Ослик. "Винни-Пух и все все все".**

Ну а теперь рассмотрим псевдокод нашей программы, чтобы понять её структуру и принципы работы.

```
for всех пикселей по оси X:
    for всех пикселей по оси Y:
        c=complex(X,y)
        for всех пикселей по оси Z:
            if abs(z)>2.0:
                break
            z=(z**2)+c
```

присвоить пикселю z чёрный цвет, если цикл не был прерван инструкцией *break* в противном случае вычислить цвет пикселя по числу итераций

Точки по краям выводимого поля выходят за пределы набора после первой итерации. Например, `abs(-2.0+-2.0j)` сразу возвращает 2.82842712475. Небольшая программа, показанная в листинге 23.1, позволит Вам проследить динамику изменения значения *z* для каждой точки в пределах выводимого поля комплексных чисел.

Листинг 23.1. Программа *ctest.py*

```
import sys
import string
```

```

c=-0.5+1j
z=0j

if len(sys.argv)>2:
    r=string.atof(sys.argv[1])
    i=string.atof(sys.argv[2])
    c=complex(r,i)

print "initial value of c", abs(c)
for i in range(64):
    print "step",i,"value of z", z, \
        "distance (abs()) of z",abs(z)
    if abs(z)>2.0:
        break
    z =(z**2)+c
    print "color z", i

```

В строках 6, 7 по умолчанию задаются те значения переменных `c` и `z`, которые мы уже использовали в предыдущих примерах. Вывод программы показан в листинге 23.1.

Листинг 23.2. Вывод программы `cctest.py`

```

initial value of c 1.11803398875
step 0 value of z 0j
distance (abs()) of z 0.0
step 1 value of z (-0.5+1j)
distance (abs()) of z 1.11803398875
step 2 value of z (-1.25+0j)
distance (abs()) of z 1.25
step 3 value of z (1.0625+1j)
distance (abs()) of z 1.45907719124
step 4 value of z (-0.37109375+3.125j)
distance (abs()) of z 3.14695655694
color z 4

```

Очевидно, что точки, сосредоточенные вокруг центра $(0,0j)$, будут окрашены в чёрный цвет, а точки по краю изображения будут окрашены в тот цвет, который мы определим как "самый далекий". Можно вывести вполне очевидный общий принцип, что по мере удаления от центра точки с большей скоростью улетают вдаль от плоскости набора Мандельброта, а точки, приближенные к центру, являются членами этого набора. Трудно сказать, действительно ли точки, не входящие в набор, удаляются в бесконечность или стремятся к какому-либо своему пределу. Тем, кому это интересно, следует заняться теорией хаоса и воспользоваться некоторыми её концепциями, в особенности концепцией аттракторов (центров

притяжения, к которым стремятся элементы стохастической системы, предоставленные сами себе). Но нас сейчас эти вопросы не интересуют.

Выбор цвета для пикселей основывается на числе итераций, необходимых для того, чтобы точка вышла за пределы набора Мандельброта. В листинге 23.2 пикселю был присвоен цвет под номером 4. Мы будем использовать те же цветовые схемы, задаваемые функциями модуля `colormap`, которые уже использовали при создании анимации в предыдущей главе. Каким будет 4-й цвет, зависит от выбранной цветовой схемы. Для функции, выбираемой по умолчанию, это будет цвет `"#F2620C"`. В листинге 23.3 показана измененная программа `cctest2.py`, которая присваивает полученный цвет атрибуту фона холста.

Листинг 23.3. Программа `cctest2.py`

```
import sys
import string
from colormap import *
from Tkinter import *

cmap = SetupColormap0(64)
c=-0.5+1j
z=0j

if len(sys.argv)>2:
    r=string.atof(sys.argv[1])
    i=string.atof(sys.argv[2])
    c=complex(r,i)

print "initial value of c", abs(c)
for i in range(64):
    print "step",i,"value of z", z, \
        "distance (abs()) of z",abs(z)
    if abs(z)>2.0:
        break
    z = (z**2) + c
    print "color z", i, cmap[i]

    root=Tk()
    root["background"]=cmap[i]
    root.mainloop()
```

Программа вычисления набора Мандельброта

Изображение на рис. 23.1 было получено с помощью программы, код которой показан в листинге 23.4.

Листинг 23.4. Программа *mandelbrot.py*

***Прим. В. Шипкова: код программы - 610 строк. Поэтому текст не привожу. Код, который был приведён здесь содержал более 100 ошибок(!!!). Длина его была 709 строк. Скачать файл можно [здесь](#). Здесь и далее ссылки на скачку одного и того же архива zip.**

Код, безусловно, слишком длинный, чтобы обсуждать его в деталях. Мы только в общих чертах рассмотрим наиболее важные части программы. На рис. 23.3 показано окно программы в том виде, в каком оно открывается сразу после запуска. Можно установить альтернативный код кнопок, как показано на рис. 23.4, если в меню Файл выбрать команду Значки на кнопках.

Класс *Msize*, определённый в строках 10–55, позволяет выбрать для показа часть изображения и устанавливать разрешение осей *X* и *Y*. В результате мы можем управлять масштабом вывода изображения. Чтобы выбрать часть изображения для просмотра под увеличением, следует воспользоваться полосами прокрутки.

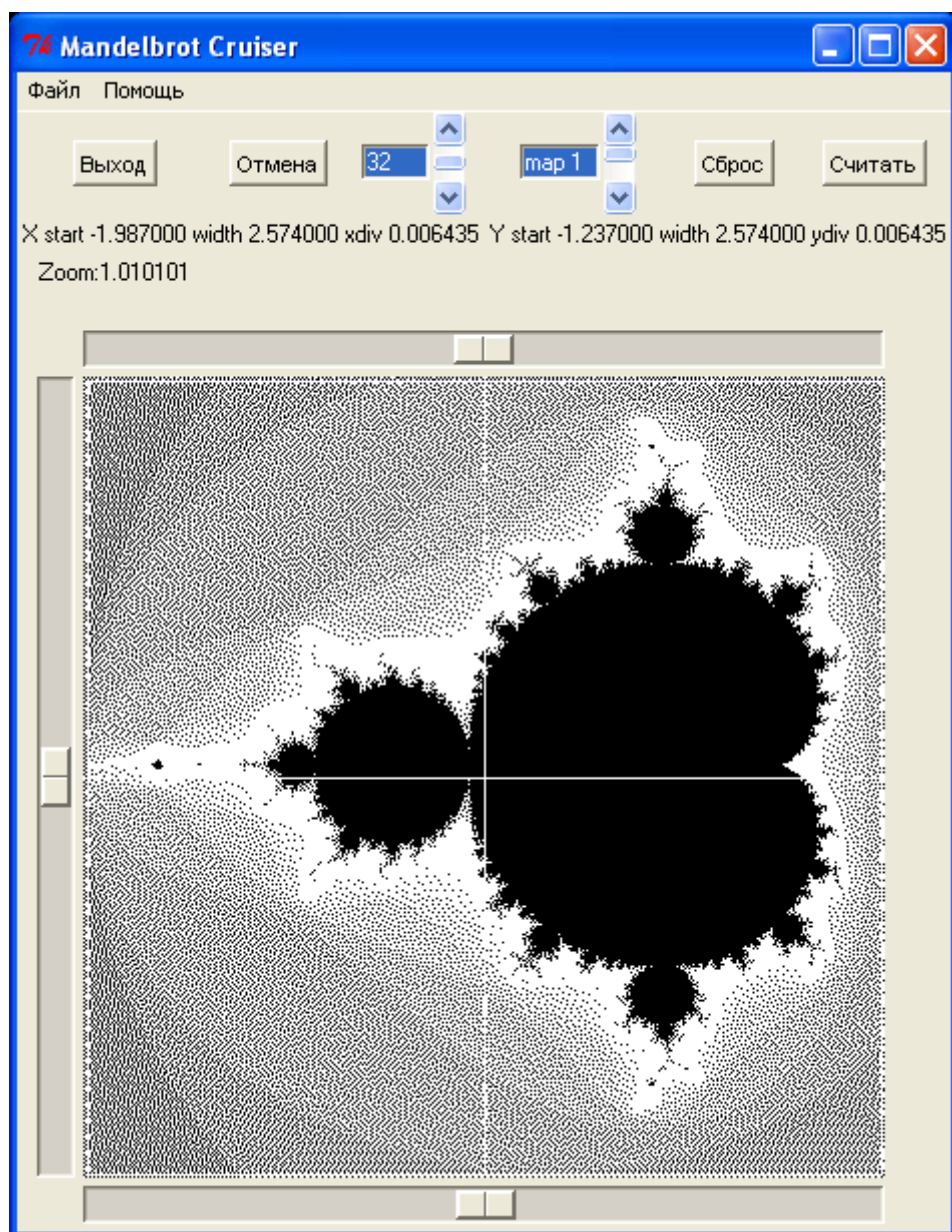


Рис. 23.3. Окно программы mandelbrot.py, которое открывается по умолчанию

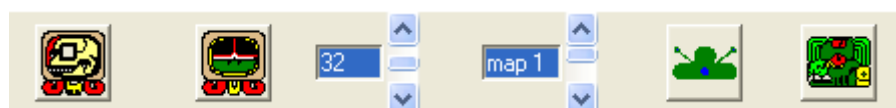


Рис. 23.4. Альтернативный вид отображения кнопок со значками

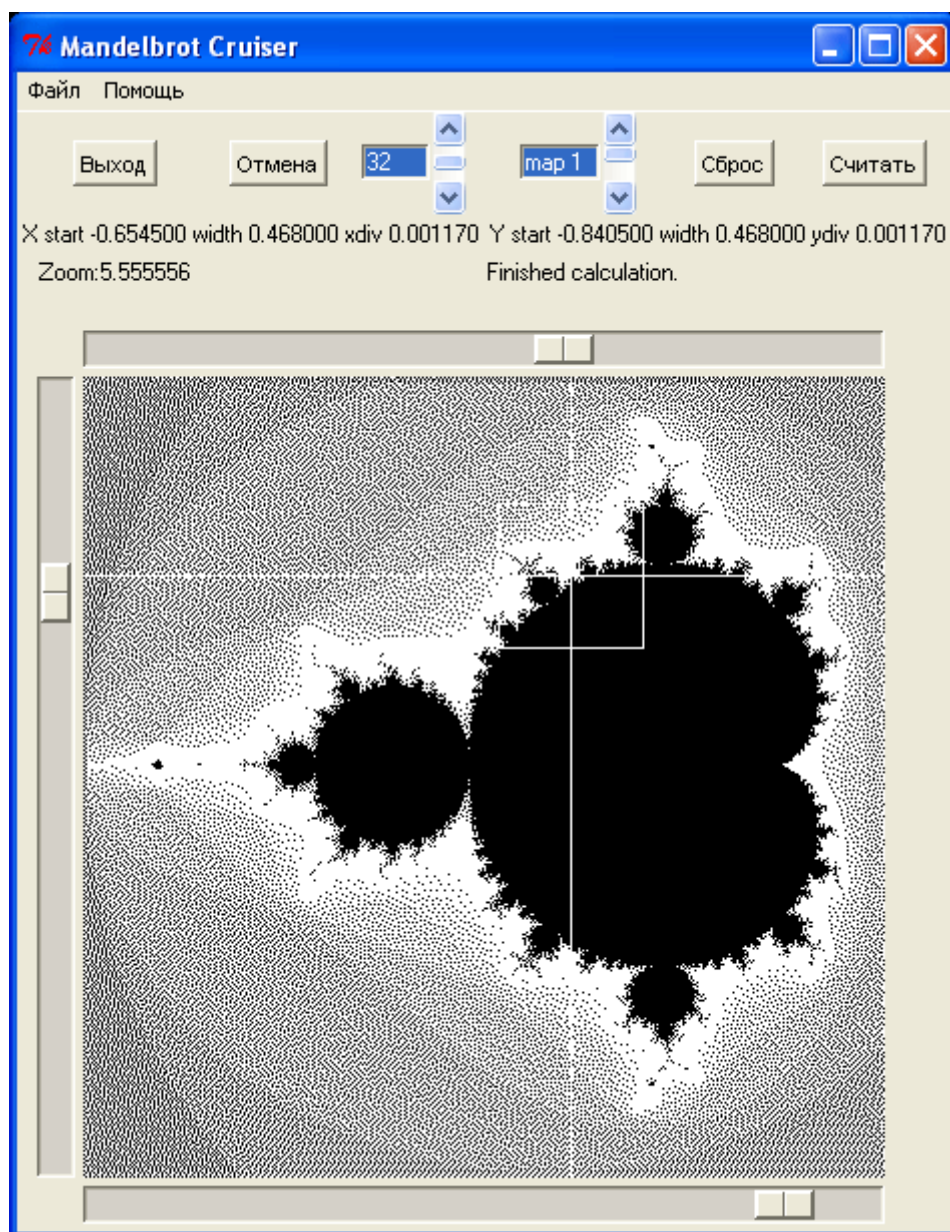


Рис. 23.5. Подготовка к увеличению в 5.55 раз

Щелкните по кнопке Считать, чтобы посмотреть выбранный участок в увеличенном виде, как показано на рис. 23.6

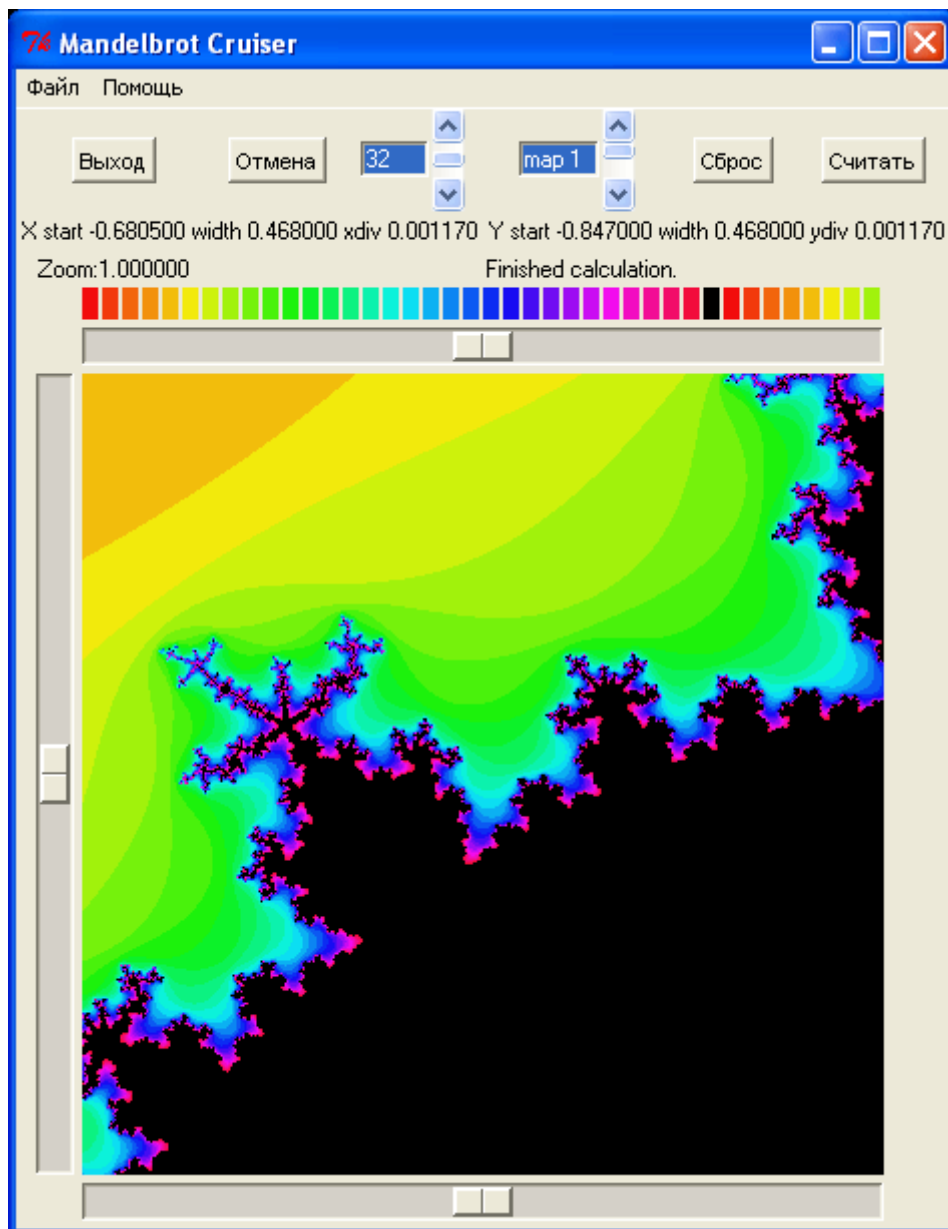


Рис. 23.6. Завершенное вычисление

Все приложение реализовано как один класс `Mandel`, что облегчает возможность его модификации с помощью наследования, например для получения наборов Джулия, которые мы рассмотрим вкратце в следующем разделе. Список `im` метода `__init__()` в строке 81 содержит имена графических файлов, используемых для быстрого вывода исходных наборов Мандельброта при запуске приложения. Если бы программа вынуждена была вычислять этот набор, то её запуск растянулся бы на несколько минут. Как Вы видели, с помощью этого изображения можно выбрать для вычисления только некоторую часть набора, чтобы сократить время и не вычислять значения для всех пикселей.

Метод `colorpoll()`, определённый в строках 363–379, обслуживает прокручиваемые списки количества цветов и цветовой схемы. Выбрав значение в списке, Вам не нужно

вводить его каким-либо способом в программу. Каждые 250 мс метод самоактивируется и считывает текущие установленные значения, которые затем передаются в переменные-члены `self.ncolor` и `self.smap`.

Методы `xliner()`, `yliner()` и `zliner()` (строки 259–346) используются для выделения части общего плана для увеличенного просмотра. Методы `xliner()` и `yliner()` прорисовывают вертикальную и горизонтальную линии в соответствии с положением бегунков на верхней и левой полосах прокрутки, отмечая в точке пересечения центр квадрата выделения. Метод `zliner()` устанавливает размер квадрата выделения по позиции бегунка на нижней полосе прокрутки.

Основная нагрузка при выполнении программы ложится на метод `calc()`, определённый в строках 399–473. При исходном разрешении 400 точек на ось этому методу приходится вычислять цвет для 160000 пикселей экрана. Для определения цвета одних пикселей достаточно одной итерации, тогда как для всех пикселей, входящих в набор, вычисление повторяется как минимум 64 раза или даже больше, в зависимости от того, какое значение присвоено переменной `dwel1`.

Большинство остальных методов класса `Mandel` достаточно просты и понятны. Хотя код в строках 517–549 требует дополнительного объяснения.

Наборы Джулия

Гастон Маурис Джулия (Gaston Maurice Julia), служивший солдатом во время Первой мировой войны, где-то незадолго до 1918 г. описал математическую матрицу, которая теперь носит его имя. Хотя Джулия был известен среди математиков своего времени, его труды оставались забытыми до тех пор, пока Беноит Мендельброт не описал свои наборы. По крайней мере частично, как признает и сам Мендельброт, он базировался на расчетах Джулия. Вас может удивить, что наборы Джулия были открыты раньше. Но ещё больше Вас должен удивить тот факт, что этот ученый вообще сумел открыть фрактальные объекты, не только не имея современного мощного компьютера, но даже не представляя, что это такое.

В чём же состоит разница между набором Мендельброта и набором Джулия. В случае с наборами Мендельброта мы последовательно выбираем все точки на плоскости, преобразовываем их координаты x и y в комплексное число и затем определяем, стремится ли эта точка улететь в бесконечность при выполнении ряда итераций базового уравнения. В случае с набором Джулия мы делаем почти то же

самое, только в качестве значения z выбираем одну точку, принадлежащую набору Мандельброта. Мы также последовательно переходим от точки к точке комплексного пространства и определяем степень стремления z к бесконечности, но значение c при этом остаётся константным.

Координаты x и y , заданные точкой c , определяют исходное значение переменной z , тогда как в наборе Мандельброта исходное значение всегда $(0, 0j)$. Таким образом, проанализировав все вышесказанное, нетрудно понять, что ту же программу, которую мы использовали для вычисления наборов Мандельброта, после небольшой модификации можно настроить на вычисление наборов Джулия. Тем более, что для выбора исходной точки c нам в любом случае нужен набор Мандельброта. На рис. 23.7 показан набор Джулия, вычисленный для исходной точки из набора Мандельброта с координатами $(-0.79, -0.15j)$.

Как и при работе с наборами Мандельброта, в окне программы мы можем выбрать часть изображения и увеличить её. Пример выбора части изображения для увеличения показан на рис. 23.8, а сам увеличенный участок на рис. 23.9.

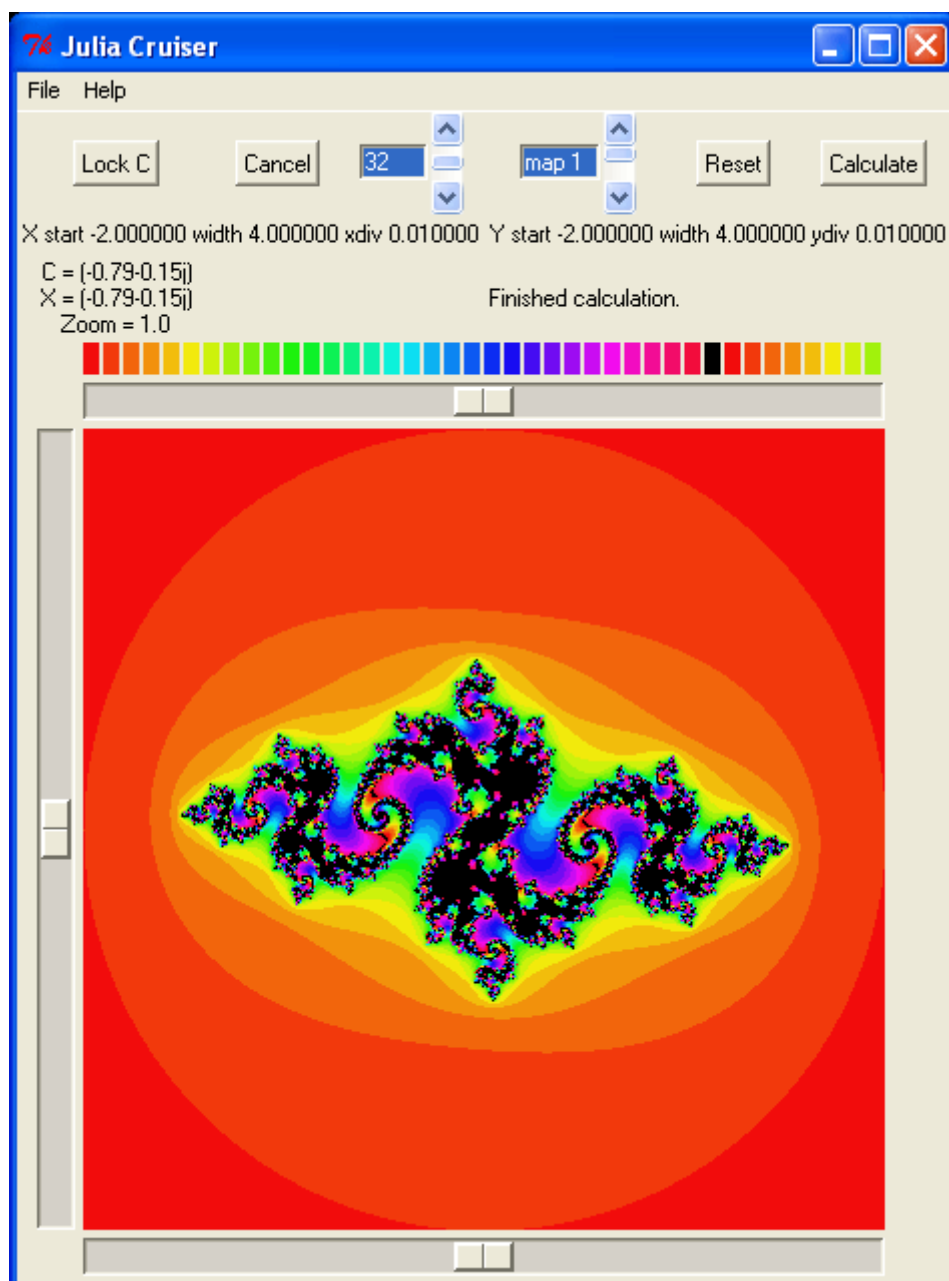


Рис. 23.7. Набор Джулия, вычисленный по точке $(-0.79, -0.15j)$

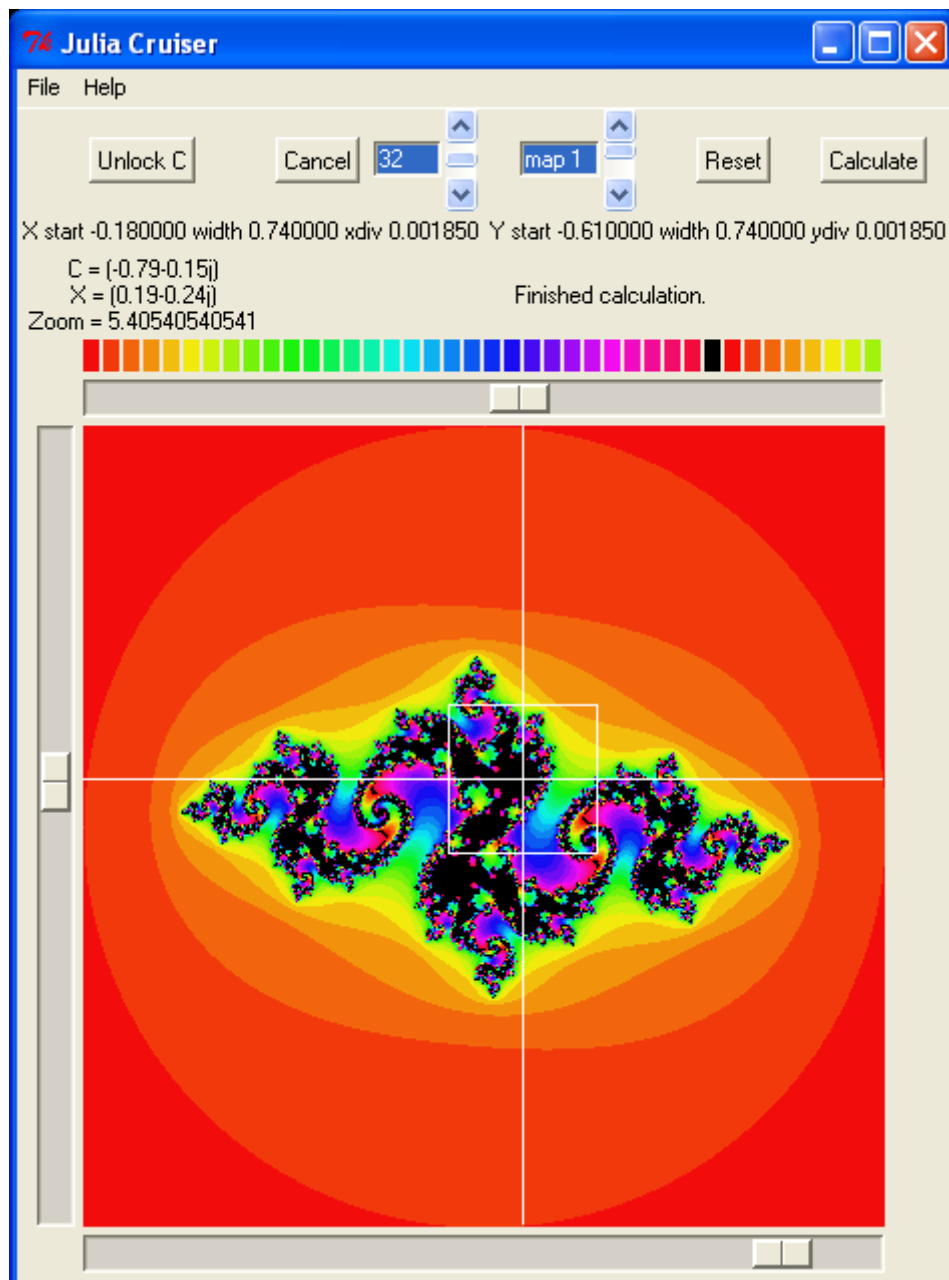


Рис. 23.8. Подготовка к увеличению

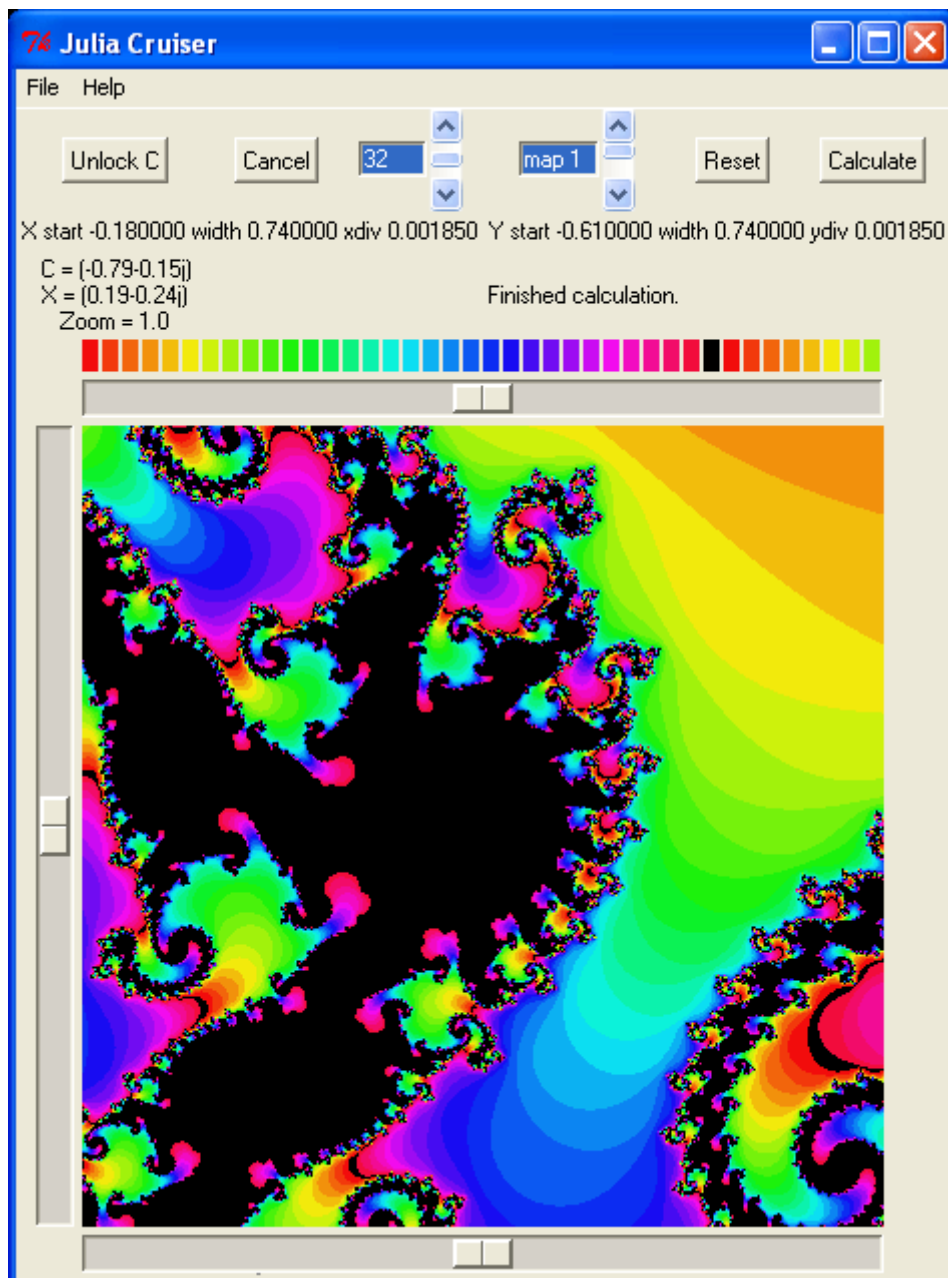


Рис. 23.9. Результат увеличения

Поскольку предыдущую программу `mandelbrot.py` мы выполнили практически полностью как один класс `Mandel`, то теперь можем унаследовать от него новый класс `Julia`. В листинге 23.5 показан код программы `Julia.py`.

Листинг 23.5. Программа `julia.py`

***Прим. В. Шипкова:** длина приведённого листинга – 269 строк. Столько же строк в оригинальном файле. Скачать файл можно [здесь](#).

***Прим. Венедикта Ли:** В этом архиве нет программы `julia.py`, а есть `julia2.py`. Её выполнение даёт те же результаты, что приведены в оригинале и показаны выше на этой странице.

Немного прокомментируем код программы `julia.py`. Главное, что нам нужно сделать в этой программе, — это изменить функциональность исходного класса `Mandel`. Это достигается замещением некоторых методов родительского класса в дочернем. Изменения коснулись методов `calc()`, `clicker()` и `porJulia()`. Эти методы теперь выполняются иначе или не выполняются вообще. Прежде всего мы изменили надпись и назначение кнопки Выход. На месте этой кнопки в окне приложения `Julia Cruiser` теперь появилась кнопка Закрепить `s`, благодаря которой закрепляются координаты точки `s`, после чего появляется возможность с помощью верхней и боковой полос прокрутки перемещать центр квадрата выделения, оставляя неизменными координаты исходной точки. Если не закрепить точку `s`, то, перемещая бегунки полос прокрутки, мы сместим исходную точку в неизвестном направлении, в результате чего не увеличим масштаб отображения, а получим новый непредсказуемый набор. Для закрепления исходной точки используется метод `lock()`, определённый в строках 32–43. Вычисление набора Джулия (строка 517 и далее в листинге 23.4) начинается с установки центральной точки и её закрепления. Собственно вычисление набора происходит в строках 529–532. (Обратите внимание, что эти строки не переписываются в программе `julia.py`, а импортируются из класса `Mandel`.)

Резюме

В данной главе Вы познакомились с наборами Мандельброта и Джулия и принципами их вычисления. Кроме того, на примерах окон приложений мы на практике применили почти все графические объекты библиотеки `Tkinter` для создания интерфейса GUI. В следующей главе мы поговорим об интерфейсе CGI, используемом при программировании для Web, рассмотрим несколько примеров программных кодов и обсудим ряд философских проблем.

Практикум

Вопросы и ответы

Помимо загрузки процессора на максимальную мощность, чем ещё полезны наборы Мандельброта?

Если Вы бесчувственны к красоте или не так чувствительны, как моя жена ("Боже, что это? Как прекрасно"), то, возможно, Вас приведёт в восторг тот факт, что фрактальная геометрия и теория хаоса используются довольно широко, начиная от прогнозирования погоды и заканчивая сжатием графических файлов.

Вы не упоминали до сих пор о фрактальной геометрии. Что это такое?

Этот термин и направление в математике были предложены Беноитом Мандельбротом. С его точки зрения фрактальная геометрия – это геометрия живой природы. Крупные объекты являются подобием мелких объектов, из которых они созданы. Представьте себе береговую линию материка, как она видна из космоса. Теперь будем увеличивать изображение, как мы это делали в окне приложения Mandelbrot Cruiser. Чем ближе мы приближаемся, тем меньшая часть береговой линии нам видна, но выглядит она все так же, как и из космоса. Тонну материалов по фрактальной геометрии и её применению Вы сможете найти в Internet. Некоторые полезные ссылки Вы найдёте ниже, в разделе "Примеры и задания".

Контрольные вопросы

1. С помощью какого уравнения вычисляются наборы Мандельброта?

- а) $E = mc^2$
- б) $a = r^2$
- в) $z = z^2 + c$
- г) $F = G(m_1 m_2)/d^2$

2. Приблизительно сколько вычислений чисел с плавающей запятой нужно выполнить для определения набора Мандельброта в поле размером 400х400 пикселей с 32 установленными цветами?

- а) 160000
- б) 1000000000
- в) 5120000
- г) 2560000

3. Как много различных наборов Джулия содержит в себе один набор Мандельброта?

- а) Один.
- б) 160000.
- в) По одному для каждой точки.
- г) Бесконечное множество.

Ответы

1. в. Наборы Мандельброта вычисляются по формуле $z = z^2 + c$.

2. г. Последнее значение – 2560000, пожалуй, ближе всего к правильному. Это не может быть число 160000, поскольку для каждой точки плоскости выполняется несколько итераций вычислений, неодинаковое для разных точек, так различно их отдаление от плоскости набора. Если это число равно 5120000, то все точки поля должны быть черными. Мы бы не стали вычислять такой набор, поскольку это скучно.

3. в. Для каждой точки набора Мандельброта можно рассчитать один набор Джулия. Такой набор Мандельброта называют каталогом наборов Джулия. Но справедливым будет также последний ответ – бесконечное множество, так как любой набор Мандельброта, по сути, содержит бесконечное число точек.

Примеры и задания

Интересную информацию, где упоминается как константа Планка, так и число гуголплекс, Вы найдёте на Web-странице по адресу <http://www.informatic.unifrankfurt.de/~fp/Tools/GetAGoogol.html>.

Дополнительную информацию о наборах Мандельброта Вы найдёте в следующих источниках.

- A. K. Dewdney. Computer Recreations. *Scientific American* 253, № 2 (August 1985), p. 16-24.
- Benoit B. Mandelbrot. The Fractal Geometry of Nature. – San Francisco: W. H. Freeman and Company, 1977, 1982, ISBN 0-7167-1186-9.
- <http://linas.org/art-gallery/escape/escape.html>.

Общепризнанной "библией" по компьютерной графике считается книга *Computer Graphics: Principles and Practice, Second Edition in C* by James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes. – New York: Addison-Wesley, 1990.

24-й час

Другие средства Python

В предыдущей главе Вы познакомились с наборами Мандельброта и Джулия, с математическими принципами их расчётов и программами вывода этих наборов на экран. В этой главе мы изучим некоторые оставшиеся средства и возможности Python. начнём мы с краткого введения программирования интерфейса CGI для Web, а затем рассмотрим некоторые

теоретические вопросы о том, как правильно писать программы на Python.

***Прим. В. Шипкова: глава достаточно условно даёт понять, что может Питон.**

Интерфейс CGI

Как уже говорилось в предыдущих главах, CGI — это интерфейс общего шлюза (*Common Gateway Interface*). Парадигмы программирования Web CGI несколько отличаются от принципов обычного интерактивного программирования. В случае программирования для Web приложения на языке Python должны размещаться в определенном каталоге, доступ к которому открыт для браузеров. Выводимые данные программа должна преобразовывать в формат HTML (*Hypertext Markup Language* — язык разметки гипертекста). В файлах формата HTML может содержаться не только текст с гиперссылками, но и заполняемые пользователем формы, данные из которых автоматически направляются одной или несколькими программам сервера, ответственным за их обработку. В этой книге не предполагалось подробно разъяснять, что такое Web-сервер, браузер, формат HTML и программирование CGI. Некоторые полезные ссылки на страницы, где эти вопросы раскрыты более глубоко, приведены в конце главы, в разделе "Примеры и задания".

На моём компьютере запущен достаточно медленный Web-сервер на базе операционной системы Windows NT 4.0 с SP5 и Web-сервер Apache 1.3.9 для Win32. Более надёжным решением было бы использование Linux вместо NT с той же версией Apache Web-сервера для Linux. Но для меня работать с NT гораздо привычнее. Услуги Web-сервера в качестве бесплатного приложения предоставляют своим клиентам многие провайдеры Internet. Правда, не многие из них поддерживают выполнение программ на языке Python. Наиболее распространённым языком программирования для Internet является Perl, но Вы можете позвонить своему провайдеру и спросить его о возможности поддержки приложений на Python. Вполне возможно, что Ваш провайдер согласится установить на своём сервере ещё один язык программирования CGI. После установки Python на сервере Вам необходимо знать, где именно он находится. По крайней мере эта информация необходима Web-серверу Apache. На моём компьютере все сценарии на языке Python помещаются в папку c:\inetpub\cgi-bin — стандартное место размещения программ CGI. Приложение Web-сервера Apache установлено в папке C:\Apache, а интерпретатор Python находится в папке C:\Python. Все

сценарии на языке Python, помещённые в папку cgi-bin для Web-сервера Apache, должны начинаться со строки `#!C:\Python\Python.exe`, иначе они не будут выполняться. В главе 15 мы рассматривали программу `fixhash.py`, которая автоматически контролирует правильность пути в первой строке всех файлов заданных каталогов. Таким образом, эта программа может оказаться теперь для Вас весьма полезной, иначе Вам вручную приходилось бы проверять все файлы программ CGI на правильность установки пути к интерпретатору.

***Прим. В. Шипкова: на самом деле приведённый ранее пример не является самым эффективным. По крайней мере - в интернете можно найти решения и по эффективней.**

Если Вы не запускаете Web-сервер на своём компьютере, а убедили в этом своего провайдера, узнайте у него, в какой папке на сервере установлен Python. Кроме того, проследите, чтобы на компьютере провайдера все стандартные модули были установлены таким образом, чтобы Python имел к ним доступ. Наиболее важным для Вас сейчас будет модуль `cgi.py`. Если Ваш провайдер использует систему Windows NT, то установить Python не составит труда. Достаточно только запустить стандартную процедуру установки и указать папку инсталляции Python. С Linux выполнить установку сложнее, но если Ваш провайдер будет чётко следовать прилагаемой к Python инструкции по установке, то проблем не должно возникнуть. Если же проблемы возникнут, то о путях их решения можно узнать на домашней странице Python или специализированной телеконференции.

В листинге 24.1 показан код традиционной программы "Hello, World!", измененной для запуска на Web-странице.

Листинг 24.1. Программа `hellocgi.py`

```
#!c:\Python\python.exe

def print_content():
    print "Content-type: text/html"
    print

def print_header():
    print "<html><title>Pythonic Hello World</title><body>"

def print_footer():
    print "</body></html>"
```

```
print_content()
print_header()
print "Hello, World!"
print_footer()
```

Я проверил эту программу на Web-серверах Apache, запущенных в системах Linux и NT. В обоих случаях программа выполнялась успешно. Убедившись в работоспособности программы, я поместил её на Web-странице англоязычного издания этой книги по адресу <http://www.pauahtun.org/cgi-bin/hellocgi.py>. Введите этот адрес в поле адреса своего браузера, чтобы увидеть ту же картинку, что показана на рис. 24.1.

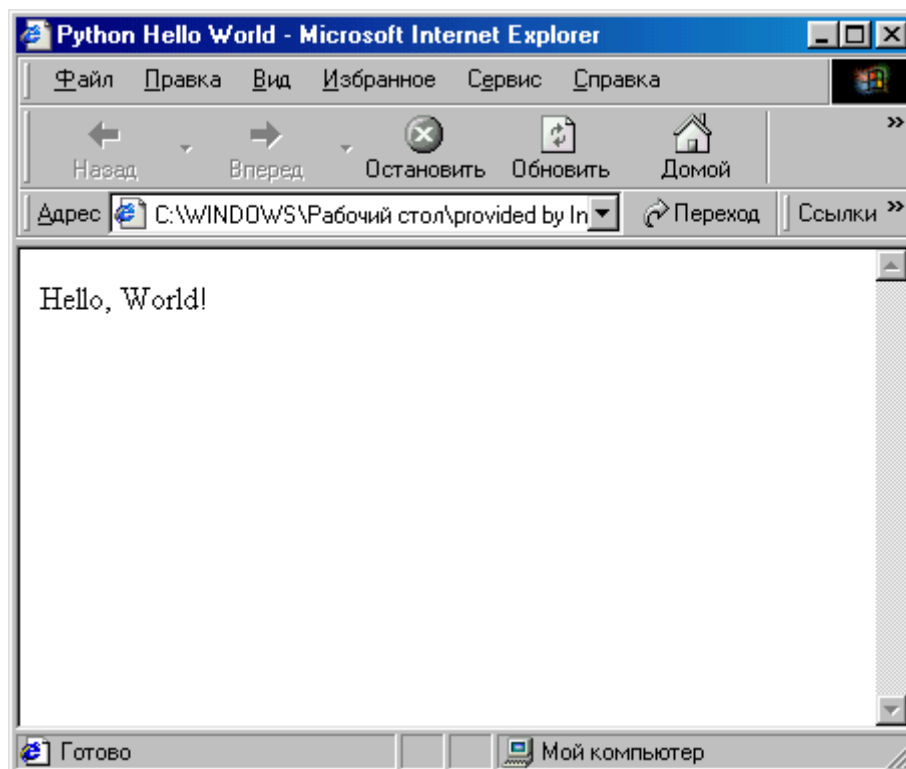


Рис. 24.1. Программа "Hello, World!" для запуска в Internet

Строка адреса состоит из зарегистрированного адреса сервера в Internet (<http://www.pauahtun.org/>), имени каталога сценариев cgi-bin и имени файла сценария hellocgi.py. Программа hellocgi.py достаточно простая. Наиболее важной частью кода является функция print_content(). Независимо от того, насколько сложным будет Ваш сценарий CGI, он всегда должен начинаться с вызова этой функции. Эта функция выводит для Web-сервера строку Content-type: text/html, которая сообщает серверу о том, что содержимое файла предназначено для показа пользователям.

Давайте создадим что-то более сложное, например форму для регистрирования программного продукта, которую пользователь сможет заполнять в окне своего броузера. После щелчка на кнопке Подача запроса программа, показанная в листинге 24.2, посылает электронное сообщение по адресу владельца формы.

Листинг 24.2. Программа register.py

```
#!/c:\Python\python.exe
import os
import sys
import cgi
import SMTP

thisscript="http://www.pauahtun.org/cgi-bin/register.py"
mailhost="mail.callware.com"
mailfrom="ivanlan@callware.com"
mailto="ivanlan@callware.com"
ccto="ivanlan@callware.com"

def print_content():
    print "Content-type: text/html"
    print

def print_header():
    print """<html><title>Pythonic Registration Form</title>
    <body bgcolor=white text=black>"""

def print_footer():
    print "</body></html>"

def print_form():
    print """
    <h1 align=center>Please Register!</h1>
    <p>Please register software before downloading it.
Thank
    you,
    %s
    <hr>
    <p>
    <form name=form action=%s method=post>
    Enter your name:
    <input type="edit" name="name"><br>
    Enter your email address:
    <input type="edit" name="email"><br>
    Enter the name of the software you are downloading:
    <input type="edit" name="software"><br>
    Enter the amount you are willing to pay in dollars for
```

```

the
    software:
    <input type="edit" name="pay"><br>
    <hr>
    <input type="submit">
    <input type="reset">
    </form>
    """ % (os.environ["SERVER_NAME"],thisscript)

print_content()
print_header()

form = cgi.FieldStorage()
if form.has_key("name") and form.has_key("email"):
    nm = form["name"].value
    em = form["email"].value
    sw = form["software"].value
    mn = form["pay"].value
    print """
        <p align=center><font size=7>Thank you!
        </font><font size=3><br>
        <hr>
        """
    print """<p align=left>Thank you %s. Your email address,
%s,
    will shortly receive an invoice for %s dollars, for
    downloading
    the %s package. If you do not pay up within 5 business
    days,
    all your files will be erased.<br>
    <hr>
    <p align=right>Have a nice day
    
    """ % (nm,em,mn,sw)
msg="""
    Hello, %s (%s):
    You recently downloaded %s from %s.
    Please send %s dollars immediately to:

    Ransom
    PO Box 6969
    Washington DC 55512

    If you do not send money, we have a catapult.
    Thank you for your attention to this matter.

    Anonymous
    """ % (nm, em, sw, os.environ["SERVER_NAME"],mn)

```

```

s=SMTP.SMTP(mailhost)
s.send_message(mailfrom, em,
    "Software Registration", msg)
s.send_message(mailfrom, ccto,
    "Software Registration", msg)
s.close()
else:
    print_form()
    print_footer()

```

Эта версия программы настроена на пересылку сообщений автору книги. Чтобы перенастроить её для себя, измените строки 7-11, подставив имя своего Web-сервера, каталога сценариев и свой адрес электронной почты. Затем введите правильный URL в поле адреса Вашего браузера, и Вы получите на экране форму, как на рис. 24.2.



Рис. 24.2. Форма, создаваемая программой register.py

Заполните форму, введя своё имя и адрес электронной почты, после чего щелкните на кнопке Подача запроса. Откроется новая страница, показанная на рис. 24.3.

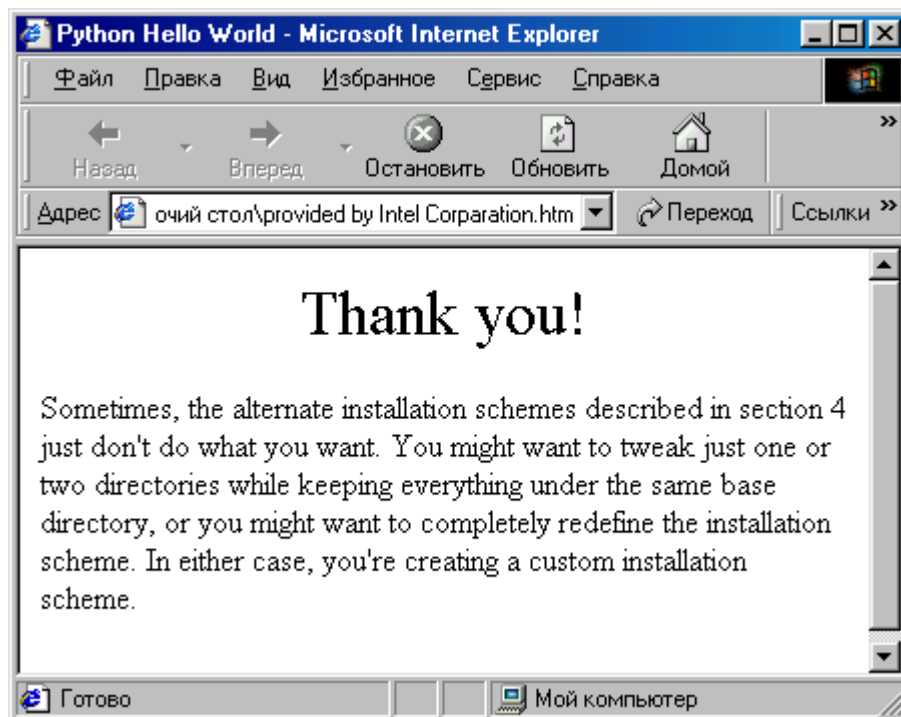


Рис. 24.3. Данные формы отправлены успешно

Но прежде чем выполнять эту программу на своём Web-сервере, нужно установить модуль SMTP.py, разработанный Владимиром Уловым (Vladimir Ulogov). Этот модуль можно загрузить с сервера, расположенного по адресу <http://starship.python.net/crew/gandalf> /SMTP.py. Модуль SMTP.py является оболочкой стандартного модуля smtpplib и существенно упрощает его использование.

Если все установки адресов в программе выполнены правильно, то она будет успешно выполняться в Linux, Windows 95/98 и Windows NT. Скорее всего, она также будет работать и в Macintosh, хотя я не испытывал этого.

Чтобы разобраться, что происходит в программе регистрации, посмотрите на строку 44 листинга 24.2. Ядро программы начинается с вывода строки типа содержимого для Web-сервера, после чего выводится строка заголовка. Эти данные остаются общими для обоих окон (см. рис. 24.2, и 24.3). В строке 48 реализуется объект класса FieldStorage из модуля cgi (этот модуль импортируется в строке 4). Программы CGI удобны для создания форм, но с их помощью сложно считывать данные, которые пользователь ввёл в поля формы на своём компьютере. Все сложности этого процесса берут на себя методы модуля cgi. В строке 49 проверяется, был ли заполнен данными объект form. Если метод has_key() возвращает false, значит, пользователь не ввёл данные в поля формы. В этом случае выполнение программы переходит к строке 86, где вызывается функция print_form(). Эта функция вновь выведет на экран пустую форму, которую Вы видели на

рис. 24.2. (Определение функции `print_form()` дано в строках 24–44.) Если же строка 49 возвращает `true`, то начиная со строки 50 происходит обработка данных, введенных пользователем. В результате обработки данных в строках 65–77 формируется сообщение, а для пользователя открывается Web-страница, (см. рис. 24.3), в которой пользователю сообщается, что его информация успешно принята. После завершения формирования сообщения для передачи в строках 79–84 создается объект SMTP, который связывается с вашим почтовым сервером и посылает копию сообщения по назначению (строки 80, 82). Соединение, открывшееся для передачи электронного сообщения, закрывается в строке 84.

Ниже, на примере листинга 24.2, представлена в общем виде последовательность задач, которые Вам нужно выполнить при создании программы CGI на языке Python.

- Импортировать модули, определить переменные и функции (строки 2–44).
- Определить функцию вывода формы с кнопкой Submit (Отправить) (строка 40) и методом `post` (строка 30).
- Обратите внимание на то, чтобы форма была передана сценарию (строки 7 и 43).
- Создать экземпляр формы (строка 48).
- Выполнить проверку заполнения формы данными пользователя (строка 49).
- Если поля формы остались пустыми, повторно вывести форму в окне браузера (строка 86).
- Если поля заполнены, считать информацию с помощью метода `cgi.FieldStorage` и сформировать сообщение (строки 50–84).
- Вывести сообщение в формате HTML об успешном приеме данных (метод `print_footer()` в строке 87).

Вооружившись этими знаниями и средствами Python, Вы сможете реализовать любую задачу по программированию интерфейса CGI.

Отладка программ

На мониторе моего компьютера расположилась парочка пластмассовых насекомых. Когда меня спрашивают, зачем тебе эти жучки, я говорю, что это мои тотемы. "А что такое тотемы?" — спрашивают меня. Тогда я объясняю, что тотемы — это головы представителей клана животных. Например, для некоторых племен индейцев Северной Америки тотемными животными были бизоны. Тотемные животные дают позволение охотникам убивать отдельных представителей клана, без которых племя не смогло бы выжить. В замен тотемные

животные требуют уважения к себе, почитания и выполнения целого ряда ритуалов, исполняемых вокруг несъедаемых частей животного, например голов. Эти ритуальные церемонии исполнялись с целью выпросить у животного прощения за то, что у него была отнята жизнь. Если охотники не следовали традициям, они могли обидеть тотемное животное, в результате чего клан мог отказаться посылать этому племени своих членов на пропитание. (Если Вы думаете, что поклонение тотемным животным ушло в прошлое с племенами североамериканских индейцев, посмотрите вокруг себя. Задумайтесь на рекламой, показываемой по телевизору, о всех этих счастливых коровках, пляшущих свинках, поющих цыплятах и прочих животных, которые из кожи вон лезут, чтобы предложить Вам: "Скушай меня!".)

Отладка программ — это работа, которая часто занимает большую часть времени работы программиста над проектом. В действительности, если бы в программах не водились "жучки", число программистов уменьшилось бы, начались бы сокращения, безработица и прочий кошмар. Вот почему у всех на виду я держу своих тотемных жучков. Я не хотел бы, чтобы клан жучков перестал посылать мне своих представителей, которых бы я отлавливал в джунглях программного кода, получая за это зарплату. Вам может показаться это глупым, кто знает? Я не обижусь. Во всяком случае, это напоминает мне, что в мире все взаимосвязано, а в программных кодах особенно.

***Прим. В. Шипкова: надо будет завести себе "жучка". ;)**

С учётом вышесказанного, простота отладки программ в Python выглядит пугающе. Какая бы проблема не возникла с программой, интерпретатор Python тут же предельно точно сообщит Вам, в какой именно строке кода произошел сбой. Кроме того, в интерпретатор встроено множество классов исключений, показывающих довольно подробные и исчерпывающие окна сообщений. Конечно, чтобы разбираться в них, Вам потребуется некоторая практика. Но допустив пару раз одну и ту же ошибку, сообщение о ней будет восприниматься Вами уже как родное.

Но, кроме того, в Python существуют специальные средства, позволяющие сделать процесс отладки программ ещё проще и нагляднее. Самым простым средством является инструкция `print`, с помощью которой из любой точки программы Вы можете выводить на экран текущие значения переменных. Для Вас существует опасность скорее недооценить, чем переоценить это средство, особенно в Python. Пожалуй, это самый эффективный по простоте скорости выполнения метод выявления сбойных строк в программе. В других языках

программирования, таких как C, где каждую новую версию программы нужно компилировать, использование функции `printf()` не столь эффективно. В Python, как только Вы ввели новую строку с инструкцией, сразу же можно выполнить код программы. Ну а добавление и удаление по ходу программы инструкций `print` не займет у Вас много времени.

Другое мощное средство — возможность выполнения программы в пошаговом режиме. Для этого программу нужно запустить на выполнение не обычным путём, а в окне специального приложения отладки. При работе с Python в режиме командной строки программа отладки вызывается командой `pdb`. Документацию по программе отладки можно загрузить с Web-страницы по адресу <http://www.python.org/doc/current/lib/module-pdb.html>. В этой документации всё достаточно подробно описано. По личному опыту работы с `pdb` я знаю, что наиболее эффективно в пошаговом режиме выполнять в программе функции и методы, чтобы убедиться, что они возвращают именно то, что требуется. Иногда вместо программы отладки также успешно можно использовать функцию `assert()`, которая запускает исключение, если принимает не то значение, которое предполагал программист. Но в других случаях программа отладки незаменима, например, когда нужно выполнить программу до вызова определённой функции, а потом продолжить выполнение в пошаговом режиме, контролируя значения всех переменных функции.

И всё же я считаю, что внимательное изучение кода самим программистом — это наиболее эффективный метод отладки программ в Python. Исследуя код, постарайтесь отвлечься от человеческой логики и представить себя на месте компьютера, выполняя шаг за шагом все строки программы. Чётко представьте себе, какое именно значение должна вернуть каждая функция и что ей для этого нужно передать. И если Вы обнаружите часть кода, вызывающую у Вас сомнение, используйте инструкцию `print`, чтобы визуализировать ваши сомнения. Не используйте одновременно слишком много инструкций `print`, иначе Вам сложно потом будет разобраться на экране, какая инструкция что выводит. Лучше всего ограничиться одной-двумя инструкциями. Если ситуация не прояснится, тогда самое время воспользоваться программой отладки, чтобы в пошаговом режиме пройти наиболее сложные участки программы. К сожалению, мы не можем позволить в себе в этой книге детально остановиться на изучении работы с программой отладки. Вам следует сделать это самостоятельно, воспользовавшись справочной системой, прилагаемой к Python.

Искусство программирования

Сейчас мы отвлечемся от изучения конкретных средств программирования и обсудим абстрактные вопросы об искусстве и красоте программирования. Слишком часто программисты при написании своих программ не задумываются о выборе элегантных решений и красоте кода. Они думают таким образом: "Кому какое дело до моего кода, главное, чтобы программа работала правильно". Другие поддерживают их: "Мы понятия не имеем, что такое красиво и что такое не красиво". Ну а кто сможет определить, что красиво и что есть искусство вообще, как научиться творить искусство? Единственный способ познать красоту — делать её. Сначала работа, потом анализ. Если Вы внимательно читали эту книгу, изучали средства программирования, выполняли упражнения, то Вам уже больше ничто не препятствует начать создавать программы прямо сейчас. В философии Дзен-буддизма об этом говорят так: "Когда Вы едите клубнику — ешьте её", т.е. наслаждайтесь вашей клубникой, вместо того чтобы ломать голову над тем, в какую тарелку её положить и какой вилкой её подцепить.

Точно так же, приступая к программированию, отвлекитесь от средств и теперь впервые задумайтесь над программой.

Интересные наблюдения были проведены мною на ежегодных конференциях майянистов (специалистов по культурному наследию индейцев Майя), проходящих в Остине (Austin), штат Техас. В течение шести дней конференции проходят практикумы для начинающих. Практикумы эти состоят в том, что новичкам раздают таблички с иероглифами Майя и предлагают перевести их. Интересно было проследить, как менялось отношение учеников к этой задаче в течение недели. Сначала ты чувствуешь себя потерянным, но видишь, что также растеряны и все окружающие тебя. К середине недели накапливается ненависть к себе и к работе. Хочется разбить вдребезги эти таблички, сесть на самолет и улететь домой. Но к концу недели наступает момент истины, когда Вы ещё не знаете, как выполнить работу, но Вас озаряет понимание того, что Вы делаете. Это переломный момент, поскольку осознание задачи позволяет найти правильные решения. Мне кажется, это очень важный момент подготовки специалиста: сместить центр внимания с обучения на самостоятельную практику, от переживаний и неуверенности — к упорному труду.

Популярность и продуктивность ежегодных рабочих встреч майянистов на протяжении вот уже 25-ти лет во многом была обусловлена убеждением основателя этих конференций Линды Шел (Linda Schele) в том, что любой начинающий может внести

свой весомый вклад в развитие науки наряду с признанными профессионалами. Многие иероглифы никогда не были бы расшифрованы, если бы не свежий взгляд новичка. С самой первой конференции Линда придерживалась принципа, что участником может стать каждый желающий. Не требовалось предъявлять справки и рекомендации, подтверждающие твою учёность в этой области. Единственным требованием был живой интерес к проблеме. Увлечённость, граничащая с наваждением, приветствовалась больше всего.

Именно такая увлечённость необходима всем тем, кто хочет овладеть программированием.

Ещё один абстрактный пример. Представьте себе, что Вы разговариваете с кем-то. Думать так: "Скорей бы он уже замолчал, чтобы я смог рассказать о себе" будет не только неприлично, но и не умно. Лучше подумайте следующим образом: "Пока я слушаю его, вместо того чтобы говорить о себе, я узнаю что-то интересное, удивительное. Мои знания расширяются". Подумайте о программировании как о Вашем диалоге с компьютером. Не задумывались ли Вы, что компьютер сможет Вас чему-то научить, если Вы прекратите хоть на минутку свой монолог.

Шанрай Судзуки (Shunryu Suzuki), один из признанных духовных лидеров Дзен-буддизма, писал в своей книге *Zen Mind, Beginner's Mind* ("Мышление Дзен, мышление новичка") следующее: "Дзен — это не вид откровения, а способ сосредоточения на ежедневных рутинных процессах. Наше понимание буддизма базируется не на интеллектуальных выкладках, а на ежедневной практике". Чуть выше я говорил Вам, что переломным этапом в обучении начинающих майянистов является "момент истины", когда студентам начинает казаться, что они понимают, над чем работают. В действительности до понимания ещё далеко, это лишь момент пробуждения сознания. Понимание наступит значительно позже, когда Вы пройдёте через сотни, если не тысячи подобных моментов истин. В действительности нет абсолютного понимания истины, поэтому и нет конца открытий, которые каждый человек делает для себя.

Много лет тому назад я пытался писать научную фантастику. Меня поражало, где "настоящие" писатели берут свои идеи. Я мучительно искал идеи, которые мог бы реализовать в своём произведении. Я записывал отдельные мысли на карточки (тогда ещё не было персональных компьютеров) и тщательно сортировал их в картотеке, но ничего хорошего из этого не получилось. Я был слишком занят поиском идей, записью их и сортировкой, поэтому на написание произведения не хватало

времени. И слава Богу, поскольку все мои идеи были старыми избитыми клише и история, которая могла бы быть написана на основе этих идей, не стоила того, чтобы её писали.

У "настоящих" научных фантастов часто спрашивают: "Где Вы берёте свои идеи?" Это, пожалуй, один из наиболее часто задаваемых вопросов. Один из авторов обычно отвечал: "Я их ворую". И этот ответ глубже, чем Вам может показаться. Это действительно то, что делаем мы все. На Земле нет ничего нового. Никакая идея не может родиться в вакууме, как элементарная частица. Поэтому никакая идея не может быть абсолютно оригинальной. Все идеи вырастают под влиянием воздействующей на нас информации из окружающего мира, т.е. под влиянием других идей. "Новая" идея — это всегда гибрид существующих концепций. Заботясь о том, что происходит вокруг Вас, фокусируя своё внимание на ежедневных рутинных процессах, Вы рано или поздно ощутите (это слово даже лучше подойдёт сюда, чем осознаете), что вокруг Вас полно задач, требующих Вашего решения. Правильнее сказать — нескончаемый поток задач. Некоторые из этих задач происходят от ваших потребностей. Их можно решить написанием небольших утилит, таких как наша программа `fixhash.py`. Другие проблемы исходят от компании, в которой Вы работаете. Для их решения Python можно использовать как совершенный язык написания прототипов программ, программ-связок и даже отдельных проектов. Но практически всегда при написании программы можно найти код другой программы, который с минимальными изменениями можно приспособить для решения ваших задач. После чего Ваш код послужит базой другому программисту для написания другой программы. Таким образом, мы все "воруем" свои коды у других, и в этом нет ничего зазорного. Напротив, это верная практика программирования, направленная на решение ежедневных задач, а не на удовлетворение собственного тщеславия.

Таким образом, обучаясь программированию, Вы проходите следующие этапы: практика без понимания; практика с пониманием целей и убежденностью в том, что если Вы постараетесь, то сможете выполнить поставленную задачу; наконец, понимание того, что ваша практика программирования — это искусство, которое и есть высшая цель. Где же сейчас на этом пути находитесь Вы? Вы достаточно напрактиковались в программировании на Python и должны быть уверены в том, что любая задача Вам по плечу. А дальше — действуйте, ешьте Вашу клубнику, не задумываясь о тарелке; катайтесь на велосипеде, не задумываясь о том, как он сохраняет равновесие на двух колесах; пишите программы, не задумываясь о средствах программирования, а просто пишите. Python — это Ваш язык, поскольку мышление Python — это

мышление начинающего. Эта мудрая истина, записанная в оригинале, показана на рис. 24.4.



Рис. 24.4. Хухуа Лин (Xuhua Lin), мастер коми-графического написания китайских иероглифов, сделал эту надпись, "Мышление Python – мышление начинающего", специально по моей просьбе для данной книги

Резюме

"На практике мышление Дзен – это мышление начинающего. Ощущение неискушенности – вот что нужно для освоения философии Дзен. Сознание начинающего пусто, свободно от привычек и опыта, готовое к овладению любой истины, к сомнению и свободному мышлению. Только в этом состоянии сознание способно принимать вещи такими, какие они есть, шаг за шагом, от открытия к открытию постигать природу всего окружающего мира. Обращение к мышлению Дзен прослеживается по ходу всей книги. Иногда прямо, иногда косвенно в каждом разделе книги подчеркивается необходимость поддержания открытости и непредвзятости своего мышления. Это древнейший способ обучения тому, как решать повседневные задачи, концентрируясь не на средствах, а на задачах".

– Ричард Бейкер (Richard Baker), предисловие к книге *Zen Mind, Beginner's Mind*.

Практикум

Вопросы и ответы

Ну и как, удалось ли Вам продать издателям хотя бы один фантастический рассказ?

Нет. Но я насобирал огромную коллекцию уведомлений об отказе, которую храню до сих пор. Одно из них самое примечательное, где на кусочке бумаги размером 2х2 дюйма просто карандашом написано "Нет".

По вашим рассуждениям можно сделать вывод, что для программиста важнее не сама программа, а процесс её написания и полученная при этом практика. Так ли это?

Так считает Будда.

Примеры и задания

Чтобы узнать больше о формате HTML, обратитесь к книге *HTML: The Definitive Guide, 3rd Edition*, by Chuck Musciano and Bill Kennedy, from O'Reilly. Также полезной будет книга *Dynamic HTML: The Definitive Reference*, by Danny Goodman, from O'Reilly.

Обширный обзор всех средств отладки программ представлен в книге *Code Complete: A Practical Handbook of Software Construction* by Steve C. McConnell. Microsoft Press; ISBN: 1556154844.

Если Вы ещё не насытились моей философской "клубникой", посетите Web-страницу по адресу <http://www.pauahtun.org/vietnam/abacus.html>.

Часть IV.

Приложения

Приложение А. Текстовые редакторы для Python

Приложение Б. Зарезервированные слова и идентификаторы в Python

Приложение В. Специальные методы классов в Python

Приложение Г. Другие ресурсы по Python

Приложение А

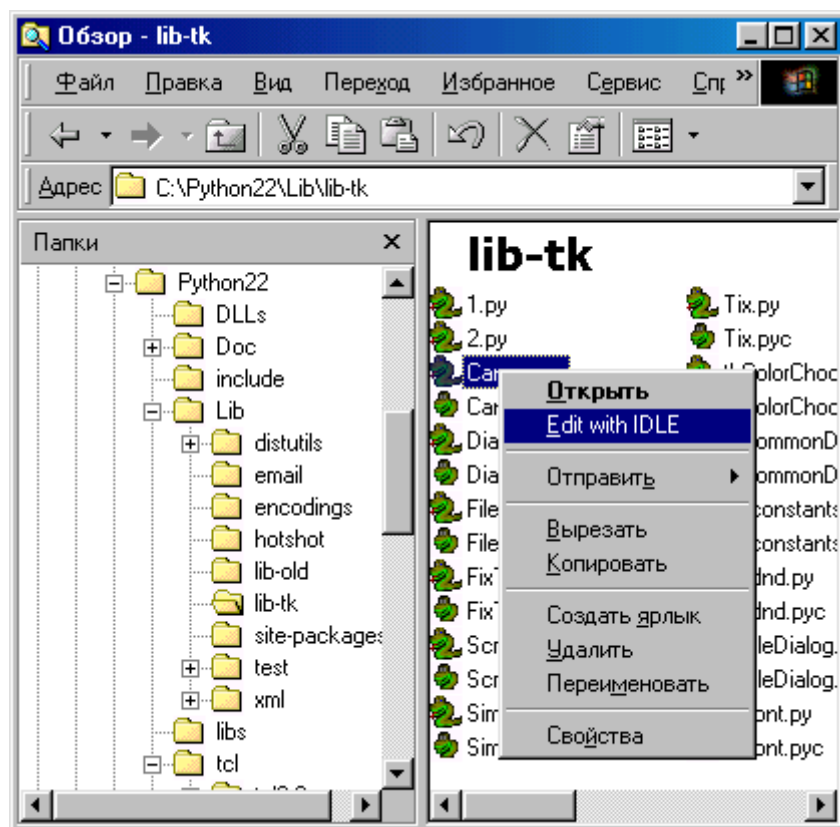
Текстовые редакторы для Python

Много полезных и удобных текстовых редакторов, пригодных для написания программ на Python, можно отыскать в Internet. Ниже представлены лишь некоторые из них.

Редактор IDLE

Редактор IDLE распространяется вместе с Python, хотя его установку можно отключить во время инсталляции Python, а во многие дистрибутивы Python для Linux он не входит. Его нет даже в дистрибутиве Python для версии Linux Red Hat, где данный язык применяется особенно широко. Дело в том, что IDLE стал использоваться для написания программ начиная с версии Python 1.5.2, а среди дистрибутивов этого языка для Red Hat особенно распространенной была версия Python 1.5.1. (Тем не менее все коды, представленные в этой книге, будут одинаково хорошо выполняться в обеих версиях.) Но системы Red hat 6.0 и 6.1 поставляются со встроенной версией Python 1.5.2 и редактором IDLE в папке /usr/bin.

Мне кажется, что редактор IDLE заслуживает того, чтобы загрузить его из Internet и установить, если у Вас на компьютере до сих пор нет этого приложения. Программа установки Python 1.5.2 в Windows берет на себя весь труд по установке самого интерпретатора, многих дополнительных модулей и библиотек, а также редактора IDLE. Единственное, чего не делает эта программа за Вас, это не выводит на рабочий стол ярлык приложения IDLE. Процедура установки Python в Windows вместе с редактором IDLE подробно была описана в главе 2. Хотя в тексте книги Вы уже видели множество окон редактора IDLE, на рис. А. 1 представлено ещё одно окно редактора с открытым файлом программы.



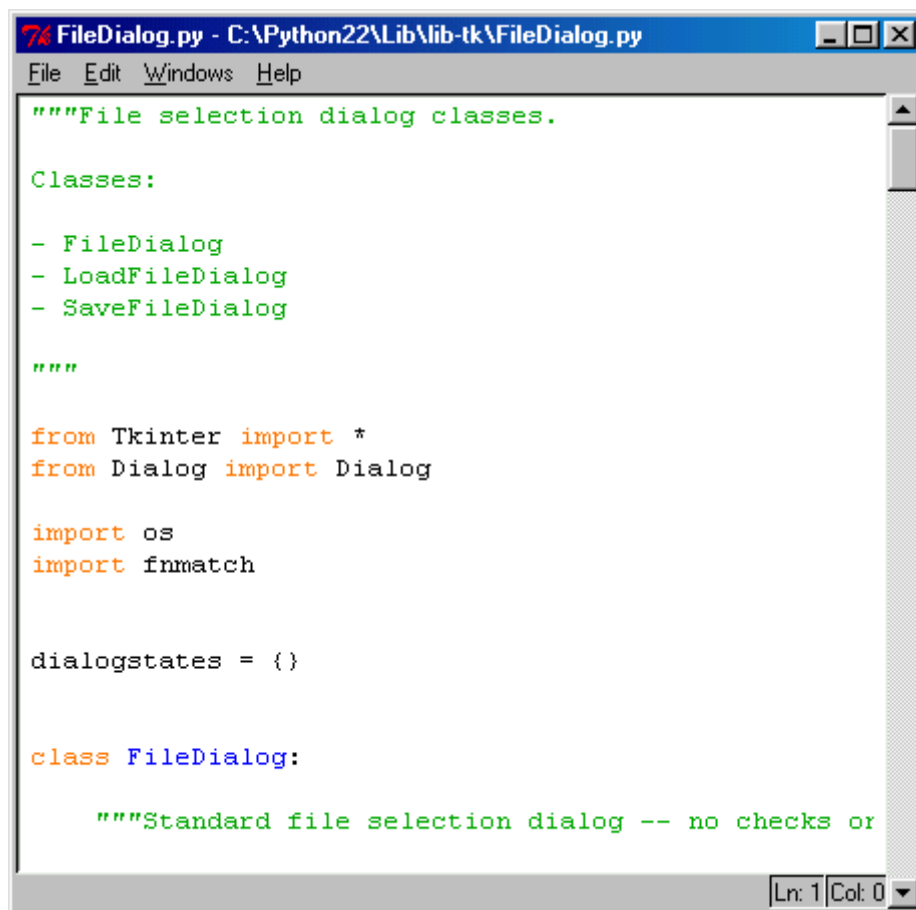


Рис. А.1. Редактирование программы на языке Python в окне IDLE

Полную информацию о Python и редакторе IDLE можно получить на домашней странице Python по адресу <http://www.python.org/>. Довольно неплохой справочник по IDLE написан Дарилом Хармсом (Daryl Harms) и представлен на Web-странице по адресу <http://www.python.org/doc/howto/idle/>. В этом справочнике подробные объяснения работы редактора сопровождаются красочными иллюстрациями.

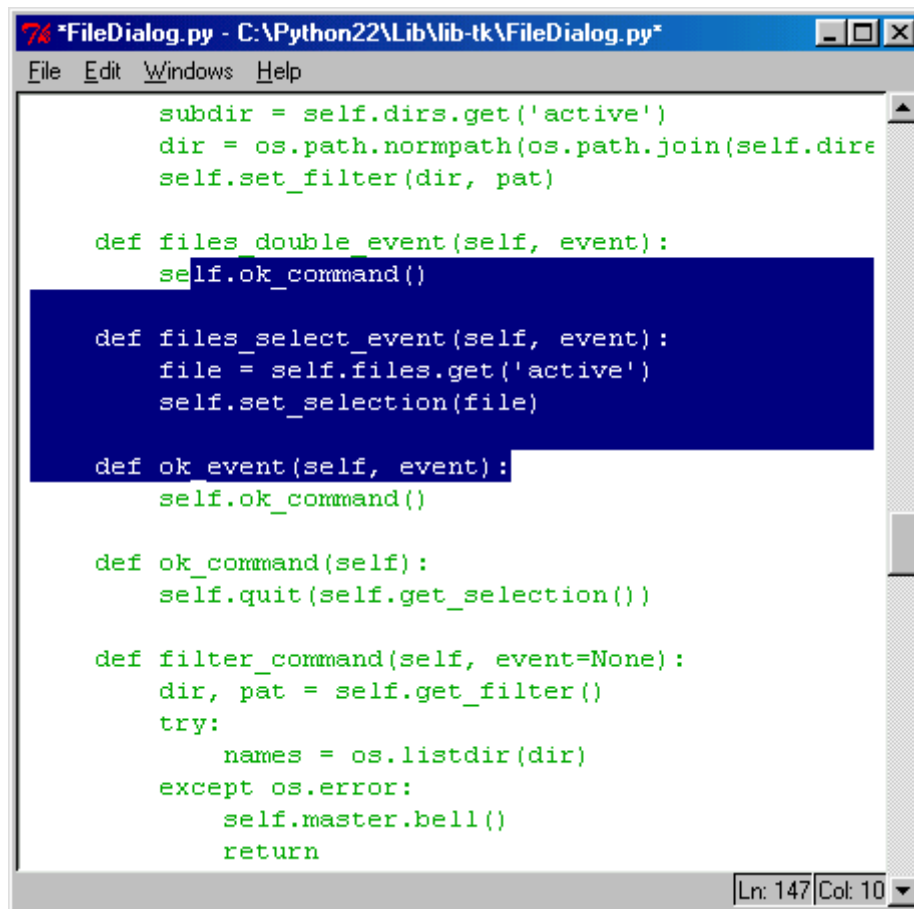
Редакторы vi и vim

Редактор vi (произносится как "ви-ай") — это классический редактор для UNIX, который затем был адаптирован для многих других операционных систем, включая Windows. На протяжении более 20 лет, с тех пор как этот редактор был разработан Биллом Джои (Bill Joy) для Berkeley UNIX, набор его базовых команд остаётся неизменным. Этот набор со временем был расширен и улучшен, но все те команды, приёмы и быстрые клавиши, которые я учил в начале 80-х, по-прежнему работают на всех платформах.

Редактор vim (*vi-improved* — улучшенный vi) представляет собой расширенную версию редактора vi. В этой версии



представлен значительно расширенный набор полезных команд, добавлено свойство синтаксического окрашивания элементов кода и установки отступов в соответствии с требованиями Python. На рис. А.2 показано окно редактора vim, в котором для редактирования открыта программа на языке Python.



```
*FileDialog.py - C:\Python22\Lib\lib-tk\FileDialog.py*
File Edit Windows Help

    subdir = self.dirs.get('active')
    dir = os.path.normpath(os.path.join(self.dirs.get('active'),
    self.set_filter(dir, pat)

def files_double_event(self, event):
    self.ok_command()

def files_select_event(self, event):
    file = self.files.get('active')
    self.set_selection(file)

def ok_event(self, event):
    self.ok_command()

def ok_command(self):
    self.quit(self.get_selection())

def filter_command(self, event=None):
    dir, pat = self.get_filter()
    try:
        names = os.listdir(dir)
    except os.error:
        self.master.bell()
    return

Ln: 147 Col: 10
```

Рис. А.2. Редактирование программы на языке Python с помощью редактора vim

Всю информацию о загрузке, установке, запуске и настройке приложения vim, а также сам продукт Вы можете бесплатно получить по адресу <http://www.vim.org/>.

Редактор Emacs

Бес сомнения, самым функциональным редактором программ среди всех поныне существующих следует признать приложение Emacs. Ниже приведена цитата из раздела "Часто задаваемые вопросы" на домашнем сервере Emacs, которая касается истории создания программы (оригинал на английском языке Вы найдёте по адресу http://www.emacs.org/FAQ/faq_4.html#SEC27).

"Название Emacs возникло как аббревиатура от Editor MACroS. Первая версия Emacs представляла собой набор макросов и была написана в 1976 г. Ричардом М. Сталлменом

(Richard M. Stallman) для редактора TECO (Text Editor and Corrector – редактор и корректор текстов, изначально Tare Editor and Corrector – редактор и корректор лент) в ITS на PDP-10. На тот момент TECO представлял собой полноэкранный редактор реального времени с перепрограммируемыми командными кнопками. Работу на Emacs в действительности начал Гай Стил (Guy Steele) с целью унифицировать наборы команд различных версий TECO, но завершил этот проект Ричард Сталлмен ".

Многие программисты утверждают, что аббревиатура Emacs произошла от названий клавиш <Esc+Meta+Alt+Control+Shift>, которые используются в этой программе в стандартном наборе командных клавиш (тогда это ещё было в новинку). Мне очень нравится Emacs. Когда-то я посвятил 6 недель своего времени изучению этого редактора и специального языкового диалекта Lisp, используемого для настройки и перепрограммирования работы Emacs в соответствии с собственными требованиями. Я всегда предпочитал работать с Emacs, а не с vi, но поскольку в последнее время мне часто приходится работать в одно и то же время с компьютерами с тремя разными операционными системами (Windows, UNIX и Linux), редактор vi оказался более переносимым, чем Emacs с моими собственными настройками.

Редактор Emacs также распространяется бесплатно. Посетите домашнюю страницу Emacs по адресу <http://www.emacs.org/>.

Редактор Xemacs

Редактор Xemacs также распространяется бесплатно. За основу при его создании была взята версия Emacs 19. Предполагалось, что он станет редактором нового поколения, унаследовавшим все лучшее от Emacs. Я не работал с этой программой, но от многих программистов слышал о нем похвальные отзывы. Лучшее место, где Вы сможете получить подробную информацию о Xemacs и загрузить версию приложения на свой компьютер, – домашняя страница редактора (<http://www.xemacs.org/>).

Приложение Б

Зарезервированные слова и идентификаторы в Python

Зарезервированные слова

В Python не допускается использование зарезервированных слов в качестве имён переменных или функций. Ниже

представлен полный список зарезервированных слов с краткими пояснениями.

Оператор	Описание
and	Логическое И
assert	Проверка истинности условия при отладке
break	Прерывание цикла while или for
class	Начало определения класса
continue	Переход к началу цикла while или for
def	Начало определения функции
del	Удаление объекта
elif	Последовательное выполнение условий if
else	Альтернативные выражения; используются с if, а также в конструкциях с try и в цикле while или for
except	В конструкции с try устанавливает блок альтернативных выражений
exec	Запуск кода Python
finally	В конструкции с try устанавливает блок выражений, выполняемых после успешного выполнения блока try
for	Начало цикла for
from	Составная часть выражения с import
global	Ссылка на глобальное пространство имён при обращении к переменной
if	Условное выражение с if
import	Поиск, чтение и выполнение кода модуля с открытием доступа ко всем переменным, функциям и классам этого модуля
in	Обращение к элементам последовательности
is	Определение запроса
lambda	Определение анонимной функции
not	Логическое НЕТ
or	Логическое ИЛИ
pass	Инструкция программе пропустить строку, ничего не выполняя
print	Вывод на стандартное устройство вывода
raise	Уведомление о возникновении исключительной ситуации
return	Выход из функции
try	Выполнение программного блока с проверкой успешности

Зарезервированные идентификаторы

Некоторые классы идентификаторов (имён переменных или функций) имеют в Python особое назначение. Ниже перечислены эти идентификаторы.

Эта переменная используется только в режиме интерактивной работы с интерпретатором. В ней сохраняются результаты последнего вычисления. Эта переменная принадлежит модулю `__builtin__`. В других режимах работы с Python эта переменная не существует, но использовать это имя в своих программах не рекомендуется. - `__*`.

Все переменные, чьи имена начинаются с одного символа подчёркивания, не импортируются из модуля инструкцией `from модуль import *`. Подобные переменные Вы можете создавать сами при разработке собственного модуля, только помните, что использовать их можно исключительно в этом модуле. - `__*`

Все переменные, чьи имена начинаются и заканчиваются двумя символами подчёркивания, такие как `__main__`, `__import__`, `__add__` и др., являются встроенными служебными членами Python. К этой группе относятся специальные методы классов, которые мы рассмотрим в приложении В. Никогда не создавайте в программах собственные переменные с такими именами. *

Два символа подчёркивания в начале имени используются для указания закрытых членов класса. Мы не изучали эту тему подробно, поскольку она выходит за рамки данной книги. Более подробную информацию Вы можете найти на Web-странице по адресу <http://www.python.org/>.

Встроенные функции

Модуль `__builtin__` содержит минимальный набор функций, имена которых недопустимо присваивать своим переменным и функциям. Эти функции можно переопределить в своей программе, но Вам лучше этого не делать. Так рекомендовано в документации на Python. Ниже представлен полный список встроенных функций. Больше информации о них Вы найдёте на домашней Web-странице Python по адресу <http://www.python.org/>.

```
__import__()  
abs()
```

`apply()`
`buffer()`
`callable()`
`chr()`
`cmp()`
`coerce()`
`compile()`
`complex()`
`delattr()`
`dir()`
`divmod()`
`eval()`
`execfile()`
`filter()`
`float()`
`getattr()`
`globals()`
`hasattr()`
`hash()`
`hex()`
`id()`
`input()`
`intern()`
`int()`
`pow()`
`ranged()`
`raw_input()`
`reduce()`
`reload()`
`repr()`
`round()`
`setattr()`
`slice()`
`str()`
`tuple()`
`type()`
`vars()`
`xrange()`

Приложение В

Специальные методы классов в Python

Специальные методы классов — это очень важное средство Python, которое делает его несравнимым по гибкости и простоте использования. Во всех пользовательских классах можно создавать собственные выполнения этих методов, которые будут автоматически вызываться интерпретатором

Python в стандартных ситуациях. Принципы использования специальных методов классов мы детально обсудили в главе 13. В этом приложении мы не будем вновь рассматривать примеры и особенности использования стандартных методов в программах. Цель данного приложения состоит в том, чтобы предоставить Вам полный список всех специальных методов классов, отсортированный по алфавиту, с кратким описанием каждого метода. Аналогичный список можно найти на сервере Python по адресу <http://www.python.org/doc/current/ref/index.html>.

Обратите внимание, что в тех случаях, когда указывается, что метод возвращает результат, необходимо создать новый объект, которому будет присваиваться результат. Например, в выражении $x = x + y$ объекты x и y являются входящими, а в результате их суммирования возвращается новый объект, который вновь присваивается переменной x .

Метод	Описание
<code>__abs__(self)</code>	Числовой метод. Возвращает абсолютное значение объекта <i>self</i> (без знака)
<code>__add__(self, other)</code>	Числовой метод или последовательности. Прибавляет <i>self</i> к <i>other</i> или выполняет конкатенацию последовательностей <i>self</i> и <i>other</i> , возвращает результат
<code>__and__(self, other)</code>	Числовой метод. Возвращает результат побитового И (оператор &) <i>self</i> с <i>other</i> и возвращает результат
<code>__call__(self[, args])</code>	Метод пользовательского класса. Если выполнение переданного класса как функции имеет смысл, использует его как метод. Список аргументов <i>args</i> не обязателен
<code>__cmp__(self, other)</code>	Метод пользовательского класса и данные других типов. Вызывается при всех операциях сравнения. Возвращает -1 если <i>self</i> < <i>other</i> , 0 – если <i>self</i> равно <i>other</i> , и 1 – если <i>self</i> > <i>other</i>
<code>__coerce__(self, other)</code>	Числовой метод. Вызывается каждый раз, когда <i>self</i> и <i>other</i> нужно привести к одному типу данных для выполнения каких-либо математических операций. Как правило, тип <i>other</i> приводится к экземпляру

	класса <i>self</i> , хотя возможно и обратное преобразование. Возвращает набор (<i>self</i> , <i>other</i>)
<code>__complex__(self)</code>	Числовой метод. Если Ваш класс может быть преобразован в комплексное число, то будет возвращен этот комплексный эквивалент
<code>__del__(seif)</code>	Метод пользовательского класса. Вызывается при удалении одного из экземпляров данного класса. При написании выполнения нужно учесть много особенностей и нюансов (см. главу 13)
<code>__delattr__(self, name)</code>	Метод доступа. Вызывается инструкцией <code>del object.name</code>
<code>__delitem__(self, key)</code>	Метод обработки последовательностей. Вызывается инструкцией <code>del object[key]</code>
<code>__delslice__(self, i, j)</code>	Метод обработки последовательностей. Вызывается инструкцией <code>del object[i:j]</code>
<code>__div__(self, other)</code>	Числовой метод. Делит <i>self</i> на <i>other</i> и возвращает результат
<code>__divmod__(self, other)</code>	Числовой метод. Делит по модулю <i>self</i> на <i>other</i> и возвращает набор из результата и остатка
<code>__float__(self)</code>	Числовой метод. Если число может быть приведено к значению с плавающей запятой, то возвращает его
<code>__getattr__(self, name)</code>	Метод доступа. Вызывается только в том случае, если обращение к атрибуту <i>object</i> , <i>attribute</i> потерпело неудачу. Возвращает значение атрибута или запускает исключение <code>AttributeError</code>
<code>__getitem__(self, key)</code>	Метод обработки последовательностей. Вызывается инструкцией <code>object[key]</code> . В качестве ключей обычно выступают целые числа. Отрицательное индексирование возможно только в том случае, если Ваш класс поддерживает его.
<code>__getslice__(self, i, j)</code>	Метод обработки последовательностей. Вызывается инструкцией <code>object[i:j]</code> .

	Оба индекса, <i>i</i> и <i>j</i> , являются целыми числами. Возможно выполнение метода с отрицательным индексированием
<code>__hash__(self)</code>	Метод пользовательского класса. Возвращает 32-разрядное значение, которое можно использовать как hash-индекс. Вызывается с использованием объекта пользовательского класса в качестве ключа словаря с помощью встроенной функции <code>hash()</code> . Для непостоянных (<i>mutable</i>) классов метод <code>__hash__</code> не вызывается.
<code>__hex__(self)</code>	Числовой метод. Возвращает строку, представляющую шестнадцатеричный эквивалент переданного класса
<code>__init__(self [,args])</code>	Метод пользовательского класса. Вызывается при реализации класса. Набор аргументов <i>args</i> не обязателен.
<code>__int__(self)</code>	Числовой метод. Если класс может быть приведён к целому числу, возвращает целочисленный эквивалент.
<code>__invert__(self)</code>	Числовой метод. Возвращает результат побитового инвертирования (оператор <code>~</code>)
<code>__len__(self)</code>	Метод обработки последовательностей. Вызывается встроенной функцией <code>len()</code> . Возвращает длину класса, как Вы её определите.
<code>__long__(self)</code>	Числовой метод. Если Ваш класс может быть преобразован в длинное целое значение, возвращает целочисленный эквивалент.
<code>__lshift__(self , other)</code>	Числовой метод. Возвращает результат сдвига влево (оператор <code><<</code>) на число бит, заданных параметром <i>other</i> , или на число единиц пользовательского класса, если это имеет смысл.
<code>__mod__(self, other)</code>	Числовой метод. Делит <i>self</i> на <i>other</i> и возвращает остаток.
<code>__mul__(self, other)</code>	Числовой метод. Умножает <i>self</i> на <i>other</i> и возвращает результат.
<code>__neg__(self)</code>	Числовой метод. Умножает Ваш класс на -1 и возвращает результат. Метод

	пользовательского класса. Возвращает 0 или 1 в зависимости от истинности проверки. Вызывается инструкцией <code>if</code> , условными выражениями и т.д.
<code>__oct__(self)</code>	Числовой метод. Возвращает строку, содержащую восьмеричное представление пользовательского класса.
<code>__or__(self, other)</code>	Числовой метод. Вызывается оператором побитового ИЛИ (<code> </code>) и возвращает результат.
<code>__pos__(self)</code>	Числовой метод. Выполняет операцию умножения пользовательского класса на 1 – операция тождества.
<code>__pow__(self, other[, modulo])</code>	Числовой метод. Возводит <i>self</i> в степень <i>other</i> . Если предоставлен аргумент <i>modulo</i> , выполняет операцию $(self**other) \mid modulo$
<code>__radd__(self, other)</code>	Числовой метод. Суммирует <i>self</i> с <i>other</i> и возвращает результат. Вызывается, например, выражением <code>1+class</code> , но не <code>class+1</code> .
<code>__rand__(self, other)</code>	Числовой метод. Возвращает результат побитового И (оператор <code>&</code>) <i>self</i> с <i>other</i> и возвращает результат, но только в том случае, если левый операнд не является членом класса <i>self</i>
<code>__rdiv__(self, other)</code>	Числовой метод. Делит <i>self</i> на <i>other</i> и возвращает результат, но только в том случае, если левый операнд не является членом класса <i>self</i> .
<code>__rdivmod__(self, other)</code>	Числовой метод. Делит <i>self</i> на <i>other</i> и возвращает набор из результата и остатка, но вызывается только в том случае, если левый операнд не является членом класса <i>self</i> .
<code>__repr__(self)</code>	Метод пользовательского класса. Вызывается встроенной функцией <code>repr()</code> и во время преобразования значения в строку (символами обратного удара). По соглашению предполагается возвращение такой строки, которая может быть

	использована впоследствии для восстановления экземпляра класса.
<code>__rlshift__(self, other)</code>	Числовой метод. Возвращает результат сдвига вправо (оператор <code>>></code>) на число бит, заданных параметром <code>other</code> , или на число единиц пользовательского класса, если это имеет смысл. Оператор вызывается только в том случае, если левый операнд не является членом , класса <code>self</code> .
<code>__rmod__(self, other)</code>	Числовой метод. Делит <code>self</code> на <code>other</code> и возвращает остаток, но только в том случае, если левый операнд не является членом класса <code>self</code> .
<code>__rmul__(self, other)</code>	Числовой метод. Умножает <code>self</code> на <code>other</code> и возвращает результат, но только в том случае, если левый операнд не является членом класса <code>self</code> .
<code>__ror__(self, other)</code>	Числовой метод. Вызывается оператором побитового ИЛИ (<code> </code>) и возвращает результат, но только в том случае, если левый операнд не является членом класса <code>self</code> .
<code>__rpow__(self, other)</code>	Числовой метод. Возводит <code>self</code> в степень <code>other</code> . Не существует r-эквивалента для операции <code>pow(self, other[, modulo])</code> . Вызывается только в случае, если левый операнд не является членом класса <code>self</code> .
<code>__rrshift__(self, other)</code>	Числовой метод. Возвращает результат сдвига вправо (оператор <code>>></code>) на число бит, заданных параметром <code>other</code> , или на число единиц пользовательского класса, если это имеет смысл. Оператор вызывается только в том случае, если левый операнд не является членом класса <code>self</code> .
<code>__rshift__(self, other)</code>	Числовой метод. Возвращает результат сдвига вправо (оператор <code>>></code>) на число бит, заданных параметром <code>other</code> , или на число единиц пользовательского класса, если это имеет смысл.
<code>__rsub__(self, other)</code>	Числовой метод. Отнимает <code>other</code> от

	<code>self</code> и возвращает результат, но только в том случае, если левый операнд не является членом класса <code>self</code> .
<code>__rxor__(self, other)</code>	Числовой метод. Выполняет операцию побитового исключающего ИЛИ (оператор <code>^</code>) и возвращает результат, но только в том случае, если левый операнд не является членом класса <code>self</code> .
<code>__setattr__(self, name, value)</code>	Метод доступа. Вызывается инструкцией <code>object.attribute = value</code> . При этом присвоенное значение добавляется в словарь экземпляра объекта и связывается с соответствующим атрибутом: <code>self.__dict__[name]=value</code> .
<code>__setitem__(self, key, value)</code>	Метод обработки последовательностей. Вызывается инструкцией <code>object[key] = value</code> . В качестве ключа <code>key</code> обычно выступает целое число.
<code>__setslice__(self, i, j, sequence)</code>	Метод обработки последовательностей. Вызывается инструкцией <code>object[i: j] = sequence</code> . Индексы <code>i</code> и <code>j</code> — всегда целые числа.
<code>__str__(self)</code>	Метод пользовательского класса. Вызывается встроенной функцией <code>str()</code> и инструкцией <code>print</code> (а также операторами <code>*</code> в выражениях форматирования). Этот метод должен возвращать строку, но к этой строке в Python нет таких требований, как к строке, возвращаемой методом <code>__repr__</code> .
<code>__sub__(self, other)</code>	Числовой метод. Отнимает <code>other</code> от <code>self</code> и возвращает результат.
<code>__xor__(self, other)</code>	Числовой метод. Выполняет побитовое исключающее ИЛИ (оператор <code>^</code>) и возвращает результат.

Приложение Г

Другие ресурсы Python

Практически все, что Вам нужно узнать о Python, можно найти на домашней странице по адресу <http://www.python.org/>.

Если Вы ищете коды Python, дополнительные модули и расширения, лучше всего начать со страницы *The Vaults of Parnassus*, расположенной по адресу <http://www.vex.net/parnassus/>. Помимо того, что на этом сервере есть традиционное средство поиска, модули в нем отсортированы по областям применения.

Средства и информация, представленные на этих двух узлах, удовлетворят практически любого, но следует упомянуть еще ряд узлов с дополнительными средствами.

Python для Atari — <http://www.gnx.com/~chrish/Atari/MiNT/python.html>.

Ресурсы Python для Macintosh — <http://www.cwi.nl/~jack/macpython.html>. На этой странице представлены средства, предназначенные исключительно для пользователей Macintosh, включая Just van Rossum's Integrated Development Environment (Интегрированная среда разработки Джаста ван Россума), куда входят текстовый редактор, программа отладки и программа просмотра классов.

Python, выполненный в Java, — <http://www.jpython.org/>.

Обучающее пособие по Python Аарона Вотерса (Aaron Walters) — <http://www.networkcomputing.com/unixworld/tutorial/005/005.html>.

На сервере *Starship* Python (<http://starship.python.net/>) предоставляется место участникам PSA (Python Software Activity) для размещения разработанных модулей и расширений, которые можно загружать бесплатно. Здесь Вы найдёте много полезных и интересных модулей.

Присоединиться к PSA можно на домашней странице этого объединения — <http://www.python.org/psa/>.

Числовой Python, Национальные лаборатории Лауренса Ливермор (Lawrence Livermore National Laboratories) — <http://xfiles.llnl.gov/python.htm>.

Python и наука — <http://www.python.org/topics/scicomp/>.

Все службы рассылки по Python можно найти на странице <http://www.python.org/psa/MailingLists.html>.

Очень интересные проекты на Python представлены на коммерческом сервере по адресу <http://www.pythonware.com/>.

Узел не англоязычных "питонистов" представлен по адресу <http://www.python.org/doc/NonEnglish.html>.

Наконец, самую свежую регулярно обновляемую документацию по Python всегда можно найти по адресу <http://www.python.org/doc/>.

***Прим. В. Шипкова: от себя ещё добавлю -**
www.zope.net.ru Официальный сайт группы добровольцев
юзающих интернет-серверZOPЕ. Это перспективная технология,
тем более, что написан этот сервер на Питоне.
[Омский http-сервер](#) Как я сказал в одном из разделов сайта
- приличная подборка софта под Питон. На том и стою.
При несильном желании в рунете ещё можно найти несколько
приличных сайтов по Питону. Но это только начало.
Python - forever!
Viva Guido van Rossum!